HZ BOOKS
华章教育

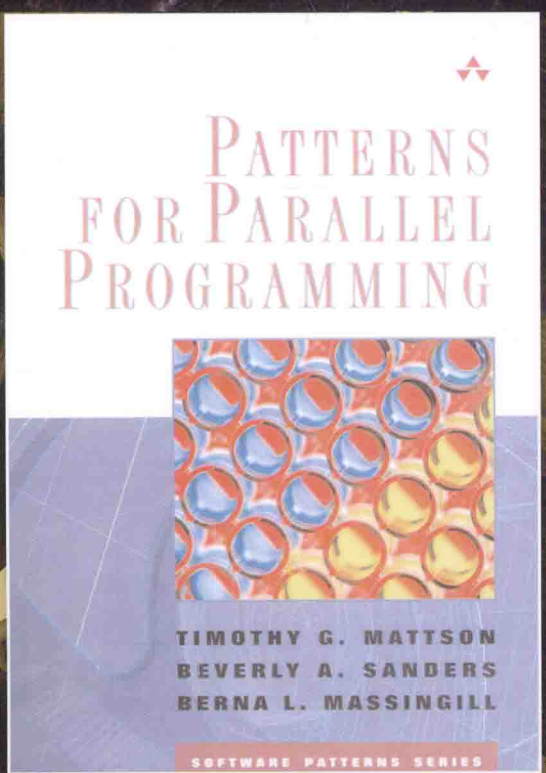
计 算 机 科 学 丛 书

PEARSON

# 并行编程模式

Timothy G. Mattson
(美) Beverly A. Sanders 著 张云泉 贾海鹏 袁良 译
Berna L. Massingill

## Patterns for Parallel Programming




机械工业出版社
China Machine Press

# 并行编程模式

## Patterns for Parallel Programming

从网格、集群到下一代游戏平台，并行计算正在成为主流。IBM、Intel、Oracle公司的超线程技术、超传输技术和多核微处理器等技术创新正在加速推动并行计算的发展。万事俱备，只欠东风——满足并行软件飞速增长需求的程序员。

本书是软件开发人员学习并行编程的权威教程，其中并没有过多讲解理论知识，而是讨论并行程序员所面临的挑战及其解决方案，并结合当前并行API的用法给出一些示例。书中引入了一种完整的、通俗易懂的模式语言，可以帮助任何有经验的开发人员编写高效的并行代码。通过学习本书，读者将意识到模式是掌握并行编程的最佳方式。本书不仅适用于高等院校计算机科学相关专业的学生，而且适用于各类软件开发人员。

### 本书主要内容包括：

- 理解并行计算和并行开发人员所面临的挑战。
- 找出软件设计中的并发问题并将其分解成并发任务。
- 管理不同任务间的数据使用。
- 生成一种可以有效利用已识别的并发性的算法结构。
- 将算法结构同需要实现的API相连接。
- 实现并行程序的特定软件结构。
- 与OpenMP、MPI和Java等当今主流的并行编程环境协同工作。

### 作者简介

**Timothy G. Mattson** 加州大学圣克鲁兹分校化学博士，英特尔生命科学社区首席发言人。他主要研究对大多程序员来说简化的并行编程技术，重点是计算生物学方面。

**Beverly A. Sanders** 哈佛大学应用数学博士，佛罗里达大学计算机信息科学与工程系副教授。她主要研究如何帮助程序员构建高质量的、正确的程序，包括形式化方法、组件系统和设计模式。

**Berna L. Massingill** 加州理工学院计算机科学博士，三一大学副教授。她的研究领域为并行式计算，以及设计模式和形式化方法。

PEARSON

www.pearson.com

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

封面设计 · 锡 彬

PEARSON

...导：计算机/并行计算

ISBN 978-7-111-49018-0

9 787111 490180 >

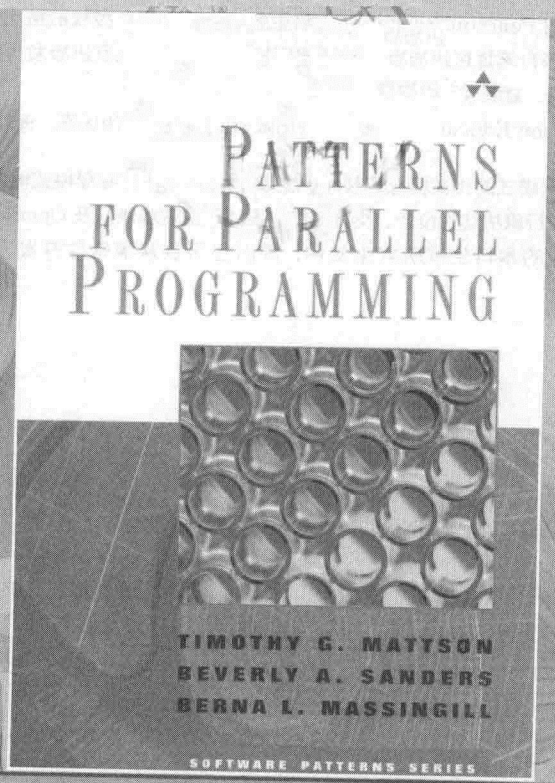
定价：75.00元

计 算 机 科 学 丛

# 并行编程模式

(美) Timothy G. Mattson
Beverly A. Sanders
Berna L. Massingill
著 张云泉 贾海鹏 袁良 译

Patterns for Parallel Programming




机械工业出版社

**图书在版编目（CIP）数据**

并行编程模式 /（美）马特森（Mattson, T. G.），（美）桑德斯（Sanders, B. A.），（美）马森吉尔（Massingill, B. L.）著；张云泉，贾海鹏，袁良译 . —北京：机械工业出版社，2014.11
（计算机科学丛书）
书名原文：Patterns for Parallel Programming

ISBN 978-7-111-49018-0

I. 并… II. ①马… ②桑… ③马… ④张… ⑤贾… ⑥袁… III. 并行程序 – 程序设计 IV. TP311.11

中国版本图书馆 CIP 数据核字（2014）第 304925 号

**本书版权登记号：图字：01-2014-6662**



本书介绍了并行编程模式的相关概念和技术，主要内容包括并行编程模式语言、并行计算的背景、软件开发中的并发性、并行算法结构设计、支持结构、设计的实现机制以及 OpenMP、MPI 等。

本书可供软件专业的本科生或研究生使用，同时也可供从事软件开发工作的广大技术人员参考。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：谢晓芳　　责任校对：殷　虹
印　　刷：北京瑞德印刷有限公司　　版　　次：2015 年 2 月第 1 版第 1 次印刷
开　　本：185mm × 260mm　1/16　　印　　张：17.25
书　　号：ISBN 978-7-111-49018-0　　定　　价：75.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

华章科技
HZBOOKS | Science & Technology

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S.Tanenbaum，Bjarne Stroustrup，Brain W.Kernighan，Dennis Ritchie，Jim Gray，Afred V.Aho，John E.Hopcroft，Jeffrey D.Ullman，Abraham Silberschatz，William Stallings，Donald E.Knuth，John L.Hennessy，Larry L.Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街 1 号
邮政编码：100037


华章教育
华章科技图书出版中心

# 译者序

Patterns for Parallel Programming

随着多核 / 众核处理器的普及以及并行计算集群应用的日益广泛，编写正确、高效的并行程序已经成为软件开发人员面临的一大挑战。并行程序可以归纳为一些具有明确定义的编程模式——用以描述并行编程的形式和方法，即并行编程模式。每一种模式都是具有相同控制结构的一类算法。并行模式的选择将直接影响并行程序的正确性和效率，从而影响整个系统的性能。因此，选择一种有效的并行编程模式对并行程序的编写及性能至关重要。

本书从寻找并发性、算法结构、支持结构和实现机制四个角度深入介绍了并行编程模式的分类、定义、实现、选择及应用。本书提供了从问题描述到最终编码的完整解决方案。首先从高层次的算法问题出发，介绍重组问题以开发算法的潜在并行性的策略及方法；在此基础上，从整体上讨论了如何利用算法的潜在并行性构造并行算法；接着从并行程序构造方法和共享数据结构的角度描述支持并行算法表达的软件构造；最后从进程 / 线程管理和进程 / 线程间交互两个方面，给出将上层空间的模式映射到特定编程环境的方法。本书不仅详细介绍了并行程序设计各个阶段的不同编程模式，讨论相应的关键技术，还详细介绍了 OpenMP、MPI、Java 这三种目前在并行编程社区常用的编程环境。此外，本书配备大量应用和程序实例，方便读者掌握相关技巧。总之，本书作者根据自己多年并行编程的实际经验，从并行算法的本质出发，以一种职业程序员易于掌握的方式对最为关键的基本知识和技术进行了细致讲解。本书可供具有一定并行编程经验的软件开发人员参考，为他们进行并行程序开发提供指导，从而降低并行编程的难度，提高并行程序的性能。

本书作者 Timothy G. Mattson、Beverly A. Sanders 和 Berna L. Massingill 从 1998 年就开始了模式语言和并行计算设计模式的相关工作，都具有丰富的并行编程经验。Timothy G. Mattson 在加州理工学院时，就将自己的分子散射软件移植到 Caltech/JPL 超立方体上。此外，Mattson 还参与了许多重要的并行计算项目，包括 ASCI Red 项目（第一个万亿次浮点运算大规模并行处理计算机）、OpenMP 开发以及 OSCAR（一种流行的集群计算包）。目前，他负责英特尔生命科学市场的战略决策，是英特尔生命科学的首席发言人。

由于时间仓促，加之书中个别术语目前没有统一译法，因此我们对一些术语采取了保留其英文名称的方法。书中翻译的错误和不妥之处，恳请广大读者不吝批评指正。

张云泉

“如果建好了它，他们就会到来。”[⊖]

我们已建成了多处理器工作站、大规模并行超级计算机和集群，但利用这些机器编程的程序员却还没有出现。少数乐于迎接挑战的程序员已经证明，大多数问题可以利用并行计算机快速求解，但普通程序员，特别是那些生活安逸的职业程序员，却忽略了并行计算机。

他们这样做十分不明智，因为并行计算机即将成为主流。多线程微处理器、多核 CPU、多处理器 PC、集群、并行游戏控制台等并行计算机正在逐步占领整个计算市场，在计算机行业中，市场上到处是这样的硬件，这些硬件唯有借助并行程序才能全速运行。但谁将编写并行程序呢？

这是一个老生常谈的问题，甚至在 20 世纪 80 年代早期“killer micros”开始取代传统向量机时，我们就十分担心如何吸引普通程序员编写并行程序。我们尝试了能想到的所有方法，包括高级硬件抽象、隐式并行编程语言、并行语言扩展和可移植消息传递库。但是经过多年的努力之后，“他们”并没有出现，绝大多数程序员并没有致力于编写并行软件。

一个常见的观点是，你不能将新技巧告诉老程序员，因此直到老程序员逐渐退出、新一代程序员逐渐成长后，并行编程问题才能够得到解决。

但我们不认同这种悲观主义态度。多年来，程序员一直在采用新的软件技术方面表现出非凡能力，许多使用 Fortran 的老程序员现在正在编写完美的 Java 面向对象程序。因此问题并非在于老程序员，而在于并行计算专家如何培养并行程序员。

这正是本书的目标，我们希望把握优秀并行程序员思考并行算法本质的过程，并以一种职业程序员易于掌握的方式讲解。为此，我们利用*模式语言*来介绍并行编程。之所以这样选择，不是因为要利用设计模式解决新领域中的问题，而是因为模式已经被证明适用于并行编程。例如，模式在面向对象设计领域非常有效，它们提供了一种用于讨论设计元素的通用语言，并且能够有效地帮助程序员掌握面向对象的设计方法。

本书包含并行编程的模式语言。前两章将介绍并行计算的一些基础知识，包括并行计算的概念和术语，而不是详尽介绍整个领域。

后 4 章介绍了模式语言，对应于创建一个并行程序的 4 个阶段。

- **寻找并发性**。识别可用的并发性，并用于算法设计中。
- **算法结构**。用一种高级结构组织一个并行算法。
- **支持结构**。将算法转化为源代码，考虑如何组织并行程序以及如何管理共享数据。
- **实现机制**。寻找特定的软件构造，实现并行程序。

这些模式紧密相关，构成了 4 个设计空间。从顶部（寻找并发性）开始，依次经历每一种模式，到达底部（实现机制）时，就可以得到并行程序的一个详细设计。

如果目标是一个并行程序，那么所需要的除了一个并行算法之外，还包括编程环境和用

---

⊖ 这是电影《梦幻之地》中的对话。影片中无法完成梦想的农场主人公，有一天听到神秘声音说：“如果建好了它，他们就会到来。”于是他铲平了自己的玉米田建造了一座棒球场，最终他的棒球偶像真的来到这里打球。——译者注

于表示程序源代码中并发性的方法。程序员过去需要面对大量不同的并行编程环境。幸运的是，随着时间的推移，并行编程社区目前主要使用三种编程环境。

- OpenMP：扩展了 C、C++ 和 Fortran，主要用来在共享内存计算机上编写并行程序。
- MPI：用于集群和其他分布式存储计算机的一种消息传递库。
- Java：一种面向对象的编程语言，支持共享内存计算机上的并行编程，并支持分布式计算的标准类库。

很多读者可能熟悉其中的一种或几种编程语言，但为了方便那些并行计算的初学者，附录简要介绍了这几种编程环境。

我们研究模式语言多年，现在将其总结为一本书以便使用。但是这并非此项工作的终点。我们期望读者有自己的新想法，并设计更好的并行编程新模式。我们可能遗漏了模式语言的某些重要特征，希望并行计算社区能推广这种模式语言。我们将继续更新并改进这种模式语言，直到它成为并行计算社区的统一观点。我们将开展一些实际的工作，例如，使用模式语言来创建更好的并行编程环境，帮助人们使用这些模式来编写并行软件。我们将一直努力，直到并行软件全面代替串行软件的那一天。

**致谢**

我们从 1998 年开始研究模式语言，从如何设计并行算法的一个模糊概念开始到完成本书，已经走过了一段漫长而崎岖的道路。如果没有以下人员的帮助，我们不可能完成这项任务。

Mani Chandy 将 Tim 介绍给 Berverly 和 Berna，并坚信我们能成为一个优秀团队。美国国家科学基金、英特尔公司、三一大学多年来一直支持本项目的研究。每年夏天在伊利诺伊举办的 PLoP 会议参与者为本书的具体并行编程模式提供了大量帮助。该会议的组织和审稿过程十分具有挑战性，但没有这些经验我们无法完成模式语言的研究。同时感谢仔细阅读本书并指出大量错误的审稿专家。

最后，还要感谢我们的家人。感谢 Beverly 的家人（Daniel 和 Steve），Tim 的家人（Noah、August 和 Martha），以及 Berna 的家人（Billie），感谢他们的支持和付出。

Tim Mattson
Beverly Sanders
Berna Massingill

### Timothy G. Mattson

Timothy G. Mattson 拥有加州大学圣克鲁兹分校的化学博士学位，研究方向为量子分子散射理论；之后在加州理工学院进行博士后研究，致力于将自己的分子散射软件移植到 Caltech/JPL 超立方体上。他曾担任多项与计算科学相关的商业和学术职务，参与了许多重要的并行计算项目，包括 ASCI Red 项目（第一个万亿次浮点运算大规模并行处理计算机）、OpenMP 开发以及 OSCAR（一种流行的集群计算包）。目前，他负责英特尔生命科学市场的战略决策，是英特尔生命科学社区的首席发言人。

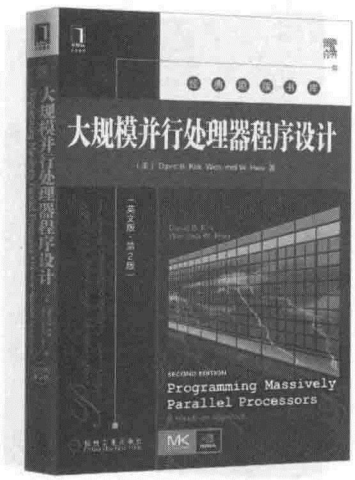
### Beverly A. Sanders

Beverly A. Sanders 拥有哈佛大学的应用数学博士学位。她曾任教于马里兰大学、瑞士联邦理工学院（ETH Zürich）以及加州理工学院，目前任教于佛罗里达大学的计算机信息科学与工程学系。她的教学和科研一直围绕着设计模式、形式方法和编程语言思想等技术的开发与应用，以帮助程序员构建高质量、正确的程序，尤其是并发程序。

### Berna L. Massingill

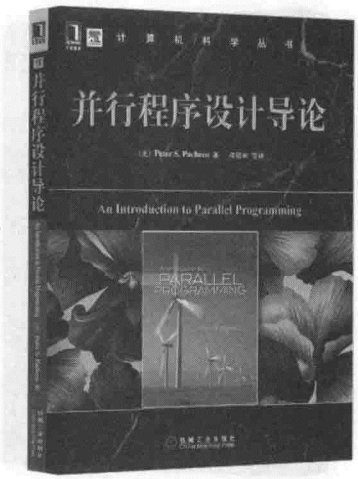
Berna L. Massingill 拥有加州理工学院的计算机科学博士学位，之后在佛罗里达大学进行博士后研究，和其他两位作者开始了关于并行计算设计模式的工作。她目前任教于三一大学（位于得克萨斯州圣安东尼奥市）计算机科学系。她拥有十余年的编程工作经验，最初从事主机系统编程工作，后来在一个软件公司担任开发人员。她的研究兴趣包括并行和分布式计算、设计模式以及形式方法。她的教学和研究目标之一是帮助程序员构建高质量、正确的程序。

# 推荐阅读



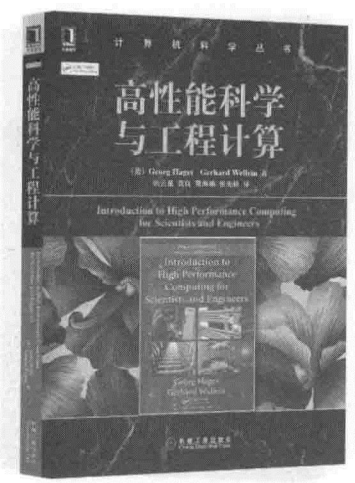

**大规模并行处理器程序设计（英文版·第2版）**

作者：David B. Kirk 等 ISBN：978-7-111-41629-6 定价：79.00元



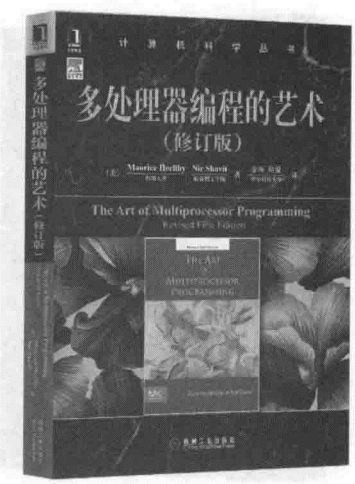

**并行程序设计导论**

作者：Peter S. Pacheco ISBN：978-7-111-39284-2 定价：49.00元




**高性能科学与工程计算**

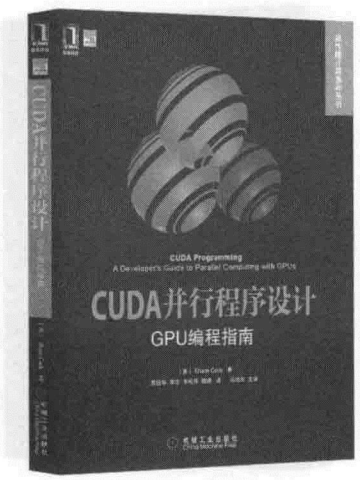
作者：Georg Hager 等 ISBN：978-7-111-46652-9 定价：69.00元




**多处理器编程的艺术（修订版）**

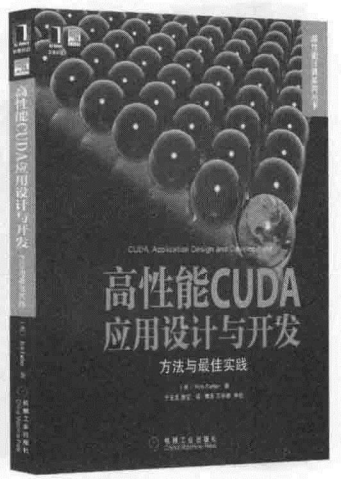
作者：Maurice Herlihy 等 ISBN：978-7-111-41858-0 定价：69.00元

# 推荐阅读



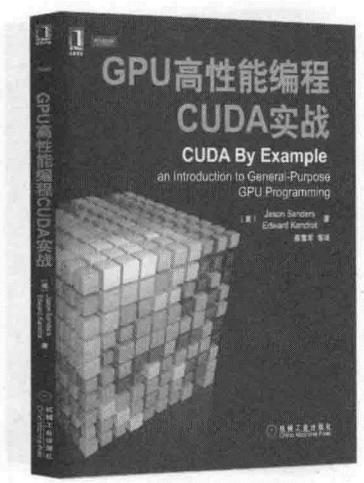

**CUDA并行程序设计：GPU编程指南**

作者：Shane Cook ISBN：978-7-111-44861-7 定价：99.00元



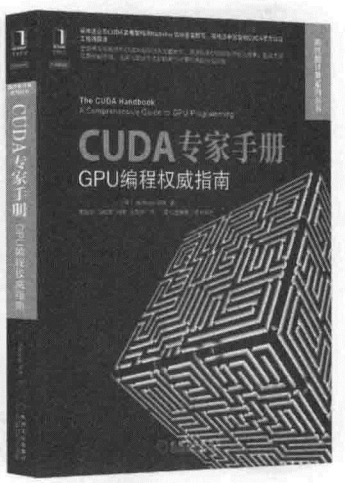

**高性能CUDA应用设计与开发：方法与最佳实践**

作者：Rob Farber ISBN：978-7-111-40446-0 定价：59.00元




**GPU高性能编程CUDA实战**

作者：Jason Sanders 等 ISBN：978-7-111-32679-3 定价：39.00元




**CUDA专家手册：GPU编程权威指南**

作者：Nicholas Wilt ISBN：978-7-111-47265-0 定价：85.00元

# 目 录

Patterns for Parallel Programming

第 1 章

Patterns for Parallel Programming

# 并行编程的模式语言

## 1.1 引言

计算机被广泛用于模拟自然科学、医学和工程学等领域中的真实物理系统，包括天气预报模拟系统、震撼电影的场景模拟系统，并且可以使用任意的计算能力来完成更加精细的模拟。无论是顾客购物模式，还是来自于宇宙的遥测数据或者 DNA 序列，它们需要分析的数据量都十分巨大。为了提供这种计算能力，设计者将多个处理单元融入一个较大的系统中。这就是所谓的*并行计算机*，它能够同时运行多个任务，在更短的时间内解决规模更大的问题。

过去，并行计算机仅用于解决一些最重要的问题，但是从 20 世纪 90 年代中期开始，随着微处理器内部开始支持多线程技术，并且在单个硅片上可容纳多个处理器内核，这种情况已经发生了根本性变化。现在，并行计算机随处可见，几乎每所大学计算机系都至少有一台并行计算机。几乎所有的石油公司、汽车制造商、药品开发公司和特效工作室都已经使用了并行计算。

例如，计算机动画的生成过程是将动画文件的信息（如光、纹理和阴影）应用于 3D 模型以生成 2D 图像，再用这些 2D 图像组成电影的帧。为较长的电影生成所需的帧时（每秒 24 帧），并行计算是非常必要的。1995 年，由 Pixar 公司发行的《玩具总动员》是第一个完全通过计算机软件制作的电影，该电影由一台包括 100 个双处理器机器的名为“renderfarm”[PS00] 的机器处理。Pixar 公司在 1999 年制作《玩具总动员 2》时利用了一台具有 1400 个处理器的系统。由于提高了处理能力，影片的效果（包括纹理、服饰和艺术效果）有大幅度提升。Monsters 公司 2001 年使用了一个拥有 250 个企业级服务器。（每个服务器包含 14 个处理器，共 3500 个处理器）的系统。然而利用更高的计算性能，包括处理器数目以及单个处理器计算性能，来提升动画效果，因此生成一幅帧需要的时间仍保持不变。



生物科学在能够从多种生物体（包括人）中获得 DNA 序列信息后实现了跨越式的发展。由 Celera 公司提出并成功使用的一种称为基因组鸟枪的排序算法，可将基因组划分为多个子段，首先通过实验确定每个子段的 DNA 序列，然后利用计算机通过重构子段间的重叠区域来重组整个序列。Celera 公司排序人类基因组时使用的计算机包括 150 台四路服务器以及一台具有 16 个处理器和 64GB 内存的服务器，实际计算包括 5 兆亿次基对基比较 [Ein00]。

SETI@home 项目 [SET，ACK+02] 的目的是寻找地外智能存在的证据，它是另一个展示并行计算能力的实例。该项目利用位于波多黎各的世界上最大的阿雷卡纳特无线望远镜搜索外太空，分析收集到的数据，搜索智能信号源。该项目的计算需求超过了世界上最大超级计算机的性能，因此只能利用*公共计算资源*满足性能需求，即将通过 Internet 互联的世界范围内的 PC 整合为一台并行计算机。项目将收集的数据划分为一些子任务，并将子任务通过 Internet 发送到每个客户端计算机上，利用这些分布的个人计算机的空闲时间进行计算。每个

客户端定期连接到 SETI@home 服务器，下载数据并执行分析计算，最后将结果发送回服务器。客户端程序被设计为一个屏幕保护程序，仅当计算机处于空闲状态时才将 CPU 贡献给 SETI 问题执行计算任务。目前，一个工作子任务平均需要一台计算机 7 ~ 8 个小时的 CPU 时间，从项目启动到现在已有 2.05 亿个工作单元得以处理。最近，也出现了一批利用公共计算资源的新项目，并且一些大公司也利用其内部个人计算机的空闲计时解决从新药筛选到芯片设计验证等问题。

尽管通过并行计算可以以更短的时间解决一些串行无法完成的问题，但实现并行也需要付出一些代价。编写并行计算软件是十分困难的，因此只有少量程序员具备丰富的并行编程经验。如果要利用并行计算机带来的并行性潜能，大部分程序员都需要学习如何编写并行程序。

本书将为有串行程序设计经验的程序员介绍编写并行程序的设计方法。虽然已经有一些优秀的书籍介绍过特殊的并行编程环境，但本书不同之处在于，我们主要介绍并行算法的构造和设计思路。因此，我们将使用*模式语言*的概念，在面向对象社区中已经大量使用了这种包含专家设计经验的高度结构化表示。

本书前两章将介绍基础知识，其中第 1 章概述理解和使用模式语言所必需的并行计算相关概念及其背景，第 2 章将更深入地讨论并行程序员所使用的基本概念和术语，其余章节介绍具体的模式语言。

## 1.2 并行编程

并行计算的关键是*可挖掘的并发性*。如果一个计算问题能够被分解成多个子问题，并且所有子问题可在相同时间内同时、安全地解决，则该问题就存在并发性。必须分析问题并发性并在代码中显示开发性，使得子问题能够并行执行，也就是说，问题解决方案必须具备并发性。

大部分大规模计算问题具备一定的并发性。程序员通过建立并行算法并在并行编程环境中实现来挖掘问题的并发性。这样，当并行程序在多处理器系统上运行时，才能在更短的时间内完成计算。此外，相对于单处理器系统，多处理器系统能处理规模更大的问题。

例如，假设计算包括求一个大型数据集的和时，串行计算将所有的值按顺序相加；如果使用多处理器，则需划分数据集，并在不同处理器上计算每个子集的和，同时完成计算，最后求所有子集之和。这样利用多处理器并行计算能更快地完成任务。若每个处理器都有私有内存，将数据分布到多个处理器上即可处理更大规模的问题。

上述简单示例展示了并行计算的本质。并行计算的目标是利用多处理器在更短时间内解决问题，以及处理规模更大的问题（与单处理所能处理的问题规模相比）。程序员需要鉴别出问题的并发性，并设计并行算法来挖掘问题中的并发性，然后在合适的并行编程环境中编写并行代码，最后在并行系统上运行。

并行编程也面临一些挑战。通常，问题所包含的并发任务间具有一定依赖性，这些子计算以不同顺序完成可能会影响程序的运行结果。例如，在上述并行求和示例中，某一子集求和计算完成后，才能与其他子集的和相加。该算法对所有任务强加了一种偏序顺序（即所有子任务完成后才能够被组合在一起计算最终结果）。此外，由于浮点运算具有非结合性和非

交换性，因此当求和运算顺序不同时，计算结果可能会有微小的数值差。设计安全的并行程序需要付出大量努力，优秀并行程序员必须非常谨慎以保证这些不确定性不会影响最终计算结果。

即使一个并行程序是“正确”的，它也可能无法通过挖掘并发性来实现性能提高。必须确保挖掘并发性而导致的额外开销不会严重影响程序运行时间，并且在其他问题中实现负载平衡通常不像求和问题那样简单。并行算法效率依赖于它与底层硬件体系结构间的映射关系，一个并行体系结构上执行效率非常高的并行算法在另一个体系结构上执行的性能可能很差。

第 2 章将更为量化地介绍并行计算并重新讨论这个问题。

## 1.3 设计模式和模式语言

设计模式描述在特定上下文中类似问题的有效解决方法。模式具有预定的格式，包括模式名称、上下文的描述、目标和限制以及相应的解决方案。其理念是记录专家经验，供他人遇到类似问题时参考借鉴。除了解决方法本身外，模式名称也是非常重要的，它构成了领域专用词汇的基础，有效加强同一领域设计者之间的交流。

设计模式最早由 Christopher Alexander 提出，其应用的领域是城市规划和建筑学[AIS77]。Beck 和 Cunningham[BC87] 最早将设计模式概念引入软件工程社区，随着 Gamma、Helm、Johnson 和 Vlissides[GHJV95] 的名为 GoF（Gang of Four 的缩写）的书籍的出版，这一概念在面向对象编程中占据了重要地位。该书收集了大量面向对象编程的设计模式。例如，Visitor 模式描述了一种组织类的方式，能独立实现不同数据结构的代码与遍历它的代码，因此遍历仅仅依赖于每个节点的类型和实现该遍历的类。这能够在不改变数据结构类的情况下灵活地添加新功能，为数据结构的不同遍历方法实现提供了便利。GoF 书中介绍的设计模式已经进入了面向对象编程词典，并已在学术文章、商业出版物和系统文档中广泛应用，这些设计模式已成为软件工程师所必需的知识。

一个组建于 1993 年的名为 Hillside Group[Hil] 的非盈利性教育组织促进了模式和模式语言的使用，更进一步地说，它鼓励人们编写通用的编程和设计实践方面的规范，因此促进了人们在计算机领域的交流。为了设计新的模式并帮助模式编写者提高技能，Hillside Group 每年举办一次编程模式语言研讨会（Pattern Languages of Programs，PLoP），并在其他地方举办了分会，例如 ChiliPLoP（在美国西部）、KoalaPLoP（在澳大利亚）、EuroPLoP（在欧洲）和 Mensore PLOP（在日本）。这些研讨会的论文集 [Pat] 包含大量模式资源，覆盖了大多数软件应用领域，并成为几本书的 [CS95、VCK96、MRB97、HFR99] 主要素材。

Alexander 最初研究模式时，不仅提出了一个模式分类方法，还提出了一种模式语言，从而引入了一种新的设计方法。在模式语言中，所有模式组织为一个特殊结构，用户遍历模式集，并选择具体模式来设计复杂系统。设计者在每个决策点上选择一个适合的模式，该模式可以导出其他多个模式，最终通过一个模式网络完成设计。因此，模式语言包括了一种设计方法学，并向应用程序开发人员提供了特定领域建议（尽管都称为语言，但模式语言并不是一种编程语言）。

## 1.4 关于并行编程的模式语言

本书给出了用于并行编程的模式语言，它具有许多优势。最直接的好处是它可以通过提

供重要问题解决方法目录、扩展的词汇表和方法学来传播专家经验。我们希望在并行程序开发的全过程中提供指导来降低并行编程的难度。程序员首先应深入理解要解决的实际问题，然后利用模式语言，最终得到详细的并行设计或代码。我们的长期目标是希望模式语言成为定性评估不同编程模型、促进并行编程工具开发的基础。

模式语言由4个设计空间组成，包括*寻找并发性*、*算法结构*、*支持结构*和*实现机制*，如图1-1所示，4个空间构成一个线性层次，其中寻找并发性位于顶部，而实现机制位于底部。

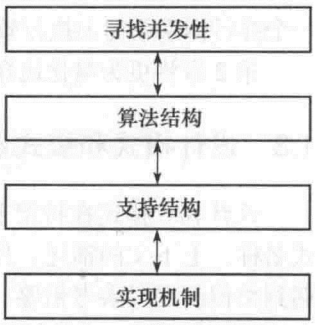


图1-1 模式语言概况

*寻找并发性*设计空间的目标是重组问题以揭示其可开发的并
5 发性。设计者在这个层次中主要面对高层次算法问题，并揭示问题的潜在并发性。*算法结构*设计空间利用潜在并发性构造算法，设计者在这个层次上考虑如何利用*寻找并发性*模式中的并发性。*算法结构*模式描述开发并发性的整体策略。*支持结构*设计空间为*算法结构*设计空间和*实现机制*设计空间提供了一个中间层。支持结构设计空间有两组重要的模式，一组程序构造方法的模式，另一组是通用共享数据结构模式。*实现机制*设计空间将上层空间的模式映射到特定的编程环境中。它描述进程/线程管理（例如，创建或销毁进程/线程）和进程/线程间交互（例如信号量、栅栏或消息传递）的通用机制。*实现机制*设计空间中的条目直接被映射到特定并行编程环境中的元素，因此不表示为模式。但它们都包含在模式语言中，
6 以便提供从问题描述到最终编码的完整解决方案。

第 2 章
Patterns for Parallel Programming

# 并行计算的背景和术语

本章将简单介绍并行编程的背景，并定义即将使用的模式中的所有的并行计算术语。由于很多术语在计算中会被反复使用，并且在不同的上下文中具有不同的含义，因此即使对于熟悉并行编程的读者，我们仍建议你浏览本章。

## 2.1 并行程序中的并发性与操作系统中的并发性

最初，在计算中挖掘程序的并发性是为了更好地利用或共享计算机中的资源。现代操作系统支持上下文切换，允许多个任务并发执行，当某个处理器因某个任务发生停顿时，可以执行其他任务。例如，这种并发性应用支持处理器通过在执行新任务而让其他任务等待 I/O 来保持繁忙。通过快速的任务切换，为每个任务分配一个处理器时间“片”，操作系统支持多个用户使用同一个系统，仿佛每个用户独占该计算机一样（尽管这样会有损系统性能）。

大多数现代操作系统能使用多个处理器增加系统的吞吐量。UNIX shell 使用并发性和称为管道的通信抽象来提供强大的模块化功能：可以编写命令来接受字节流作为输入（消费者）和产生字节流作为输出（生产者）。多个命令通过管道将命令的输出和下一个命令的输入串联起来，利用这些简单的块可以构建一些复杂的命令。每个命令在各自的进程中执行，而所有的进程并发执行。因为对于如果管道中没有可用的缓存空间则生产者会阻塞，如果数据不可用则消费者会阻塞，这极大地简化了对命令之间计算结果流移动的管理工作。最近，Windows 操作系统允许用户同时处理多个事件，网络经常出现明显的 I/O 延迟现象，并且几乎每一个 GUI 程序都具有并发性。



在并行程序和操作系统中，尽管在安全处理并发性方面的基本概念是一致的，但存在着一些重大的区别。对于操作系统而言，问题不是寻找并发性——并发性是操作系统固有的特性，管理一组并发执行的进程集（代表用户、应用程序以及诸如打印池的后台操作），并提供同步机制以确保资源安全共享。尽管如此，操作系统的并发性必须具有健壮性和安全性：进程之间互不干涉（有意或者无意的），且整个系统不应该因为单个进程出现故障而崩溃。在并行程序中，需要考虑进程之间的相互独立性（而操作系统不需要），寻找和挖掘并发性非常具有挑战。性能目标对于操作系统和并行程序有所不同。在操作系统中，性能目标通常与吞吐量或响应时间相关，宁愿牺牲一些效率来获得健壮性和资源分配的公平性。而对于并行程序而言，目标是最小化单个程序的执行时间。

## 2.2 并行体系结构简介

并行体系结构的种类有几十种，包括工作站网络、商用 PC 集群、大规模并行超级计算机、紧耦合对称多处理器和多处理器工作站。本节将概述这些系统，主要介绍与程序员相关的一些特点。

### 2.2.1 Flynn 分类法

到目前为止，描述这些体系结构最通用的方式是使用 Flynn 分类法 [Fly72]。他根据指令流数目和数据流数目对所有的计算机进行分类，其中，流是计算机操作的指令序列和数据序列。利用 Flynn 分类法，有 4 种类型：SISD、SIMD、MISD、MIMD。

**单指令单数据**（SISD）：在一个 SISD 系统中，一个指令流处理一个数据流，如图 2-1 所示。这是最常见的冯·诺依曼模型，几乎应用于所有单处理器计算机中。

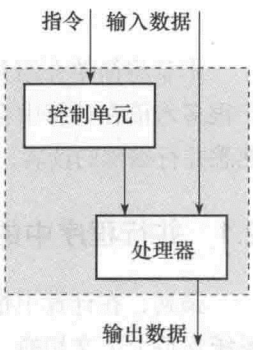


图 2-1 SISD 体系结构

**单指令多数据**（SIMD）：在一个 SIMD 系统中，一个指令流被并发地广播到多个处理器上，每个处理器拥有各自的数据流（如图 2-2 所示）。Thanking Machines 和 MasPar 等最初的系统属于 SIMD。CPP DAP Gamma II 和 Quadrics Apemille 是最近的实例，这些机器通常是针对专门的应用而部署的（例如，数字信号处理），它们适用于细粒度并行，并且进程间通信较少。向量处理器（以流水线的方式操作向量数据）也属于 SIMD，挖掘这种并行性的工作通常由编译器来完成。

8

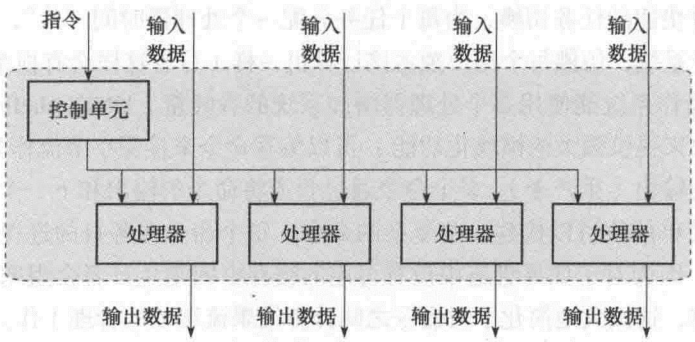


图 2-2 SIMD 体系结构

**多指令单数据**（MISD）：没有知名的系统属于此类型，提到它仅出于完整性的缘故。

**多指令多数据**（MIMD）：在一个 MIMD 系统中，每个处理单元拥有自己的指令流，并且这些指令流操作自己的数据流。如图 2-3 所示，由于其他的每一种体系结构都能映射到 MIMD 体系结构上，因此该体系结构是最通用的体系结构。绝大多数的现代并行系统都属于此类型。

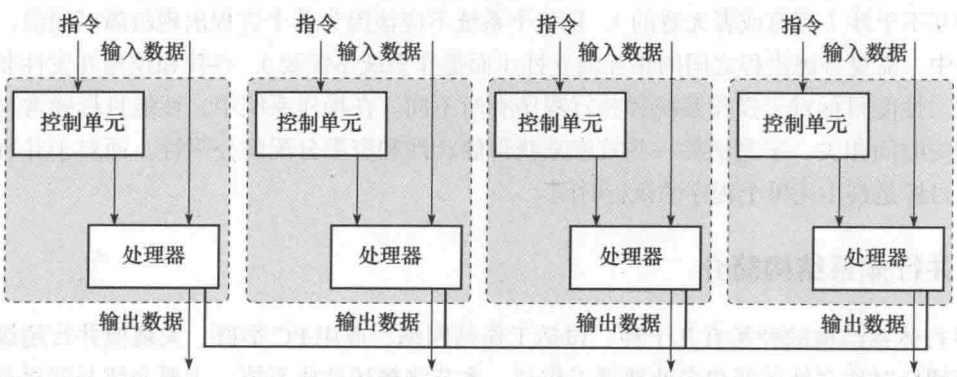


图 2-3 MIMD 体系结构

### 2.2.2 MIMD 的进一步分类

在 Flynn 分类法中，MIMD 类别太宽泛而用途不大。MIMD 类别通常根据内存组织方式进一步划分。 9

**共享内存**

在一个共享内存系统中，所有的进程共享同一地址空间，并且通过读写共享变量进行通信。

共享内存系统中有一类称为 SMP（对称多处理器）。如图 2-4 所示，所有的处理器通过连接共享一块公用的内存，并以相同的速度访问内存中所有位置。对于编程而言，SMP 系统被认为是最简单的并行系统，因为程序员不用在处理器间分配数据结构，处理器 / 内存带宽是一个典型的限制因素。所以，SMP 系统可扩展性较差，并且局限于处理器数量较少的系统。

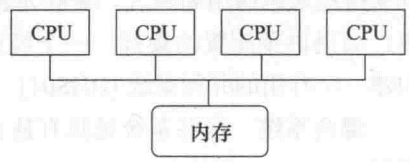



图 2-4 SMP 体系结构

其他共享内存系统类型主要称为 NUMA（Non-Uniform Memory Access，非均匀内存访问）。如图 2-5 所示，内存是共享的，即所有的处理器都可以访问该内存，但是内存中某些内存块的物理位置相较于其他处理器的距离更接近于其所关联的处理器。这降低了内存的带宽瓶颈，允许系统拥有更多的处理器。尽管有上述好处，但也带来了处理器访问内存的时间上的重大的差异，依赖于内存位置与处理器的“远近”。为了缓解非均匀访问的影响，每个处理器配置了一个缓存（cache），以及保持缓存一致性的协议。因此，这些体系结构的别称是缓存一致非统一内存访问系统（ccNUMA）。逻辑上讲，在 ccNUMA 系统上编程类似于在 SMP 上编程。但为了获取最优性能，程序员更加需要关注数据本地化问题和缓存效果。

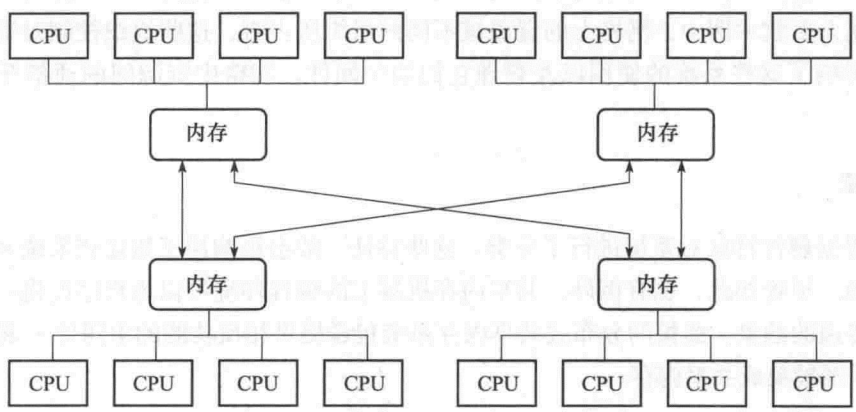



图 2-5 NUMA 体系结构示例 10

**分布式内存**。在分布式内存系统中，每个进程拥有自己的地址空间，并且通过消息传递（发送和接收消息）与其他进程进行通信。分布式内存计算机的示意图如图 2-6 所示。

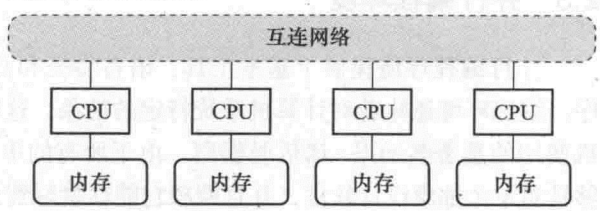



图 2-6 分布式内存体系结构

由于通信速度依赖于处理器互连的拓扑结构和技术，因此通信速度相差很大，最快几乎相当于共享内存（紧密集成的超级计算机），最慢到几个数量级差别（例如，通过以太网互接的 PC 集群）。程序员编写程序时必须显式地管理处理器间的通信和数据分配。

分布式内存计算机传统上分成两类：MPP（大规模并行处理器）和集群。在 MPP 中，处理器与网络基础结构紧耦合，并专用于并行计算机。这些系统具有很强的可扩展性，在某些示例中，单个系统能够支持数以千计的处理器 [MSW96,IBM02]。

集群是通过商用网络将商用计算机互连起来组成的分布式内存系统。计算机是运行
11 Linux 操作系统的 PC，这样的集群称为 Beowulf 集群。伴随着商用互连网络的改良，该类系统变得越来越通用和强大。集群为获取并行计算能力提供了廉价的组织方式 [Beo]。目前，很多厂商提供预配置的集群。一个经费有限的团队甚至可以利用过时和废弃的 PC 组成的集群构建一个有用的并行系统 [HHS01]。

**混合系统**。这些系统是具有独立地址空间的节点集群，每个节点包含几个共享内存的处理器。

根据 van der Steen 和 Dongarra 的“Overview of Recent Supercomputers”[vdSD03]，该文对当前以及将要出现的商用超级计算机进行了简要的描述，通过快速网络互连的 SMP 集群所组成的混合系统是当前高性能计算主要趋势。例如，在 2003 年年底，世界上最快的 5 台计算机中有 4 台是混合系统 [Top]。

**网格**。网格是通过 LAN 和 / 或 WAN 将分布的和异构的资源进行互连而构成的系统 [FK03]。通常互联的网络是 Internet。网格的设计初衷是连接多个超级计算机来解决更大规模的问题，因此它可以看作分布式内存或者混合 MIMD 计算机的一种特殊类型。最近，网格计算的思想演变成了一种共享异构资源的常用方式，比如，计算服务器、内存、应用服务器、信息服务，甚至科学仪器。网格与集群的不同之处在于网格中的各种资源不需要一个公用的管理点。在大多数示例中，网格上的资源被不同的组织所占有，这些组织控制对资源的使用策略。这影响了这些系统的使用以及管理它们的中间件，网格中资源间的通信开销最值得探讨。

### 2.2.3 小结

我们根据硬件特征对系统进行了分类。这些特征一般会影响用于描述该系统并发性的原始编程模型。尽管如此，也有例外，共享内存机器上的编程环境可以为程序员提供分布式内存和消息传递的抽象。虚拟的分布式共享内存系统包含提供相反功能的中间件：将分布式共享内存的机器抽象成共享内存。

## 2.3 并行编程环境

并行编程环境提供了基本工具、语言特征和应用程序编程接口（API）用于构建并行程序。编程环境意味着对计算机系统特定的抽象，这种抽象称为“编程模型”。传统的串行计算
12 机使用的是著名的冯 · 诺依曼模型。由于所有的串行计算机使用该模型，因此软件设计者能够针对单个抽象设计软件，并且期望它能够映射到大多数（并非全部）串行计算机上。

遗憾的是，并行计算存在很多可能的模型，反映处理器互连构成的并行系统的不同方式。

最通用的模型是基于广泛应用在并行体系结构上的模型：共享内存、基于消息传递的分布式内存，或者这两种的混合。

编程模型与某种并行系统紧密相连，导致程序在并行计算机之间不具有可移植性。由于软件的生命周期长于硬件，一些组织机构拥有多种类型的并行计算机，大多数程序员认为编程环境应该能够让他们编写的并行程序具有可移植性。此外，显式管理并行计算机中大量的资源是非常困难的，因此，对并行计算机进行更高层次的抽象是非常有用的。20 世纪 90 年代中期，出现了大量的并行编程环境。表 2-1 列出了其中一些部分。这给应用程序开发这带来了巨大的困扰，并且阻碍了将并行计算用于主流应用程序中。

表 2-1 20 世纪 90 年代中期的一些并行编程环境

| "C* in C | CUMULVS | Java RMI | P-RIO | Quake |
|---|---|---|---|---|
| ABCPL | DAGGER | javaPG | P3L | Quark |
| ACE | DAPPLE | JAVAR | P4-Linda | Quick Threads |
| ACT++ | Data Parallel C | JavaSpaces | Pablo | Sage++ |
| ADDAP | DC++ | JIDL | PADE | SAM |
| Adl | DCE++ | Joyce | PADRE | SCANDAL |
| Adsmith | DDD | Karma | Panda | SCHEDULE |
| AFAPI | DICE | Khoros | Papers | SciTL |
| ALWAN | DIPC | KOAN/Fortran-S | Para++ | SDDA |
| AM | Distributed Smalltalk | LAM | Paradigm | SHMEM |
| AMDC | DOLIB | Legion | Parafrase2 | SIMPLE |
| Amoeba | DOME | Lilac | Paralation | Sina |
| AppLeS | DOSMOS | Linda | Parallaxis | SISAL |
| ARTS | DRL | LiPS | Parallel Haskell | SMI |
| Athapascan-Ob | DSM-Threads | Locust | Parallel-C++ | SONiC |
| Aurora | Ease | Lparx | ParC | Split-C |
| Automap | ECO | Lucid | ParLib++ | SR |
| bb_threads | Eilean | Maisie | ParLin | Sthreads |
| Blaze | Emerald | Manifold | Parlog | Strand |
| BlockComm | EPL | Mentat | Parmacs | SUIF |
| BSP | Excalibur | Meta Chaos | Parti | SuperPascal |
| C* | Express | Midway | pC | Synergy |
| C** | Falcon | Millipede | pC++ | TCGMSG |
| C4 | Filaments | Mirage | PCN | Telegraphos |
| CarlOS | FLASH | Modula-2* | PCP: | The FORCE |
| Cashmere | FM | Modula-P | PCU | Threads.h++ |
| CC++ | Fork | MOSIX | PEACE | TRAPPER |
| Charlotte | Fortran-M | MpC | PENNY | TreadMarks |
| Charm | FX | MPC++ | PET | UC |
| Charm++ | GA | MPI | PETSc | uC++ |
| Chu | GAMMA | Multipol | PH | UNITY |
| Cid | Glenda | Munin | Phosphorus | V |
| Cilk | GLU | Nano-Threads | POET | ViC* |

（续）

| "C* in C | CUMULVS | Java RMI | P-RIO | Quake |
|---|---|---|---|---|
| CM-Fortran | GUARD | NESL | Polaris | VisifoldV-NUS |
| Code | HAsL | NetClasses++ | POOL-T | VPE |
| Concurrent ML | HORUS | Nexus | POOMA | Win32 threads |
| Converse | HPC | Nimrod | POSYBL | WinPar |
| COOL | HPC++ | NOW | PRESTO | WWWinda |
| CORRELATE | HPF | Objective Linda | Prospero | XENOOPS |
| CparPar | IMPACT | Occam | Proteus | XPC |
| CPS | ISETL-Linda | Omega | PSDM | Zounds |
| CRL | ISIS | OOF90 | PSI | ZPL |
| CSP | JADA | Orca | PVM | |
| Cthreads | JADE | P++ | QPC++ | |

幸运的是，20 世纪 90 年代后期，并行编程行业统一到两个主流的并行编程环境：用于共享内存系统的 OpenMP [OMP] 和基于消息传递的 MPI [Mesb]。

OpenMP 是一组语言扩展，通过编译器指令实现。目前该实现支持 Fortran、C 和 C++。OpenMP 被频繁地用于在串行代码中增加并行性。通过在循环附近添加编译器指导语句，例如，编译器能够产生代码用于并行执行循环中的迭代。编译器负责大多数线程创建和管理的细节。OpenMP 程序在 SMP 机器上工作效果很好，但是由于底层的编程模型没有包含非统一内存访问时间的概念，因此不太适用于 ccNUMA 和分布式内存机器。

MPI 是一组例程库，提供进程管理、消息传递和一些集合通信操作（对程序中所有进程进行操作，例如，栅栏、广播和归约）。由于程序员负责数据分布和使用消息在进程间显式通信，因此编写 MPI 程序并非易事。由于该编程模型假定系统是分布式内存类型，因此，MPI 对于 MPP 和其他分布式共享内存的机器是个不错的选择。

无论是 OpenMP 还是 MPI，都不适合于由多个多处理器节点组成的更大的混合体系结构（该系统中每个节点拥有多个进程和一个共享内存，节点之间的地址空间是独立的）：OpenMP 模型无法识别非统一内存访问时间，因此其数据分配在非 SMP 机器上性能低下，而 MPI 不包含用于管理驻留于共享内存中数据结构的概念。混合模型的一种解决方法是：在每个共享内存节点中使用 OpenMP，节点之间使用 MPI。这种方式能够很好地工作，但是要求程序员在单个程序中使用两种不同的编程模型。另外一种选择是在共享内存和分布式内存上都使用 MPI，放弃共享内存编程模型的优势，即便硬件支持该模型。

13
~
14

能够简化可移植性并行编程和更准确地反映底层并行体系结构特点的新的高级编程环境是目前研究的主题 [Cen]。商业领域中比较常用的做法是扩展 MPI 和 OpenMP，在 20 世纪 90 年代中期，MPI 论坛定义了称为 MPI 2.0 的 MPI 扩展，尽管该实现在本书编写之际还没有得到广泛的使用。它是 MPI 的一个非常复杂的扩展，包括动态进程创建、并行 I/O 和其他特性。对于在现代混合体系结构上编写程序的人们而言，他们特别感兴趣的是 MPI 2.0 中的单边通信。单边通信模仿了共享内存系统的一些特性，允许一个进程写入或读取其他进程的内存区域。术语“单边”是指读或写由初始化进程发起，没有显式地包括其他参与的进程。对单边通信更为复杂的抽象可以在 Global Arrays[NHL96、NHK+02、Gloa] 软件包中获得。Global

Arrays 与 MPI 一起工作，帮助程序员管理分布式的数组元素。程序员定义了数组和其在内存中的放置之后，程序通过执行 MPI 中“puts”或者“gets”函数来操作数组，不需要显式地管理哪个 MPI 进程“拥有”数组的特定部分。本质上，全局数据为全局共享的数组提供了抽象。这仅限于数组，数组是并行计算中非常通用的数据结构，尽管有局限性，但是非常有用。

就像扩展 MPI 来模仿共享内存环境的某些优点一样，OpenMP 也被扩展成运行于分布式共享内存环境。每年的 WOMPAT（研讨 OpenMP 应用程序和工具）研讨会上有许多探讨在集群和 ccNUMA 环境中的各种方法与经验的论文。

MPI 被实现成例程库，以便被使用串行编程语言编写的程序调用，而 OpenMP 是串行编程语言的一组扩展。它们代表了并行编程环境中两种可能的类别（库和语言扩展），并且这两种环境考虑了绝大多数并行计算。尽管如此，还存在另一种并行编程环境类型，即语言本身拥有并行编程特性，Java 就是这样的语言。Java 最初并不是用于支持高性能计算的，它是一种面向对象、通用的编程环境，拥有显式地表达共享内存中并发处理的特性。此外，标准的 I/O 和网络软件包使 Java 在机器间的进程之间通信变得非常容易，这使得在共享内存和分布式内存模型中编写程序变为可能。最新的 java.nio 包支持 I/O 操作，在某种程度上使程序员不太方便，但能带来巨大的性能提升。Java 2 1.5 包含了对并发编程的新支持，主要位于 java.util.concurrent.* 包中。此外，还有很多支持不同并行计算方法的包。

尽管存在其他通用编程语言，无论是先于 Java 出现还是最近出现的（例如 C#），其中都包含了用于表达并发性的构造，Java 是首先被广泛使用的语言。因此，许多程序员使用 Java 初次尝试并发和并行编程。尽管 Java 在软件工程方面提供了很多优势，但是目前 Java 并行程 15
序的性能难以与科学计算应用中的 OpenMP 或者 MPI 程序匹敌。Java 的设计在这些领域中也存在几种重大缺陷（例如，强调可移植性和可重现结果的浮点模型，而不是尽可能挖掘可用的浮点硬件能力；低效的数组处理和缺少处理复数的便利机制）。随着 Java 编译器技术的提升、新的软件包和语言扩展的出现，Java 和其他编程语言之间的性能差异正在逐步缩小，尤其对于符号或者其他非数值问题。Titanium 项目 [Tita] 就是 Java 语言针对 ccNUMA 环境中高性能计算而设计的。

本书中，我们选择了 OpenMP、MPI 和 Java 三种环境，将在示例中使用它们——选择 OpenMP 和 MPI 是因为它们具有普遍性，而选择 Java 是因为许多程序员用它来进行并发编程的初次尝试。它们的概述见附录。

## 2.4 并行编程术语

本节定义了模式语言中常用的术语。其他术语的定义可以在术语表中找到。

**任务**。设计并行程序的第一步是将问题分解成多个任务。一个任务是一个指令序列，并且作为一个小组（group）一起操作。这个小组相当于一个算法或者程序的某个逻辑部分。例如，考虑两个 $N$ 阶矩阵相乘。根据算法的构造方式，任务可以为：1）矩阵子块相乘，2）对矩阵的行和列进行内积，或者 3）矩阵相乘中单个循环迭代。这些都是在矩阵相乘中定义任务的合理方式，也就是说，对任务的定义反映了算法设计者对问题的思考。

**执行单元**（UE）。任务需要映射到某个 UE，例如，进程或者线程，才能执行。进程是使程序指令执行的资源集合。这些资源包括：虚拟内存、I/O 描述符、运行时栈、信号处理程序、

用户和组 ID 以及访问控制令牌。更高层级的观点是：*进程*是一个“重量级”执行单元，它拥有独立的地址空间。*线程*是现代操作系统中的基本 UE。线程与进程相关，共享进程的环境。这使得线程具有轻量性（也就是说，线程之间的上下文切换耗费的时间很少）。更加高级的观点是：线程是一个“轻量”UE，它和进程中的其他线程共享一块地址空间。

我们将使用*执行单元*作为可能并发执行的实体集合中某个元素的通用术语，通常是进程
16 或者线程。当进程和线程之间的差异不太重要时，这种定义方式有利于早期的程序设计。

**处理单元**。我们使用术语处理单元（PE）表示执行指令流的硬件部件。硬件单元是否可以称为 PE 取决于上下文。例如，一些编程环境将 SMP 工作站集群中的每个工作站看作单个指令流的执行单位，在这种情形下，PE 是一个工作站。但是，不同的编程环境运行在同一硬件上时，可能将每个工作站中的每个处理器看作单个指令流的执行单元，在这种情况下，PE 是单个处理器，每个工作站包含多个 PE。

**负载均衡和负载均衡方法**。为了执行并行程序，任务必须映射到 UE，然后 UE 再映射到 PE。映射方式对并行算法的整体性能有重大的影响。避免出现部分 PE 执行绝大多数任务，而其他 PE 处于闲置状态，这是至关重要的。*负载均衡*（Load balance）是指有效地在 PE 之间分配任务。*负载均衡方法*（load balancing）是将任务分配到 PE 上的过程，要么是静态的要么是动态的，从而使任务分配尽可能均匀。

**同步**。在一个并行程序中，由于任务调度和其他因素的不定性，计算中的事件可能不会一直以相同的顺序重现。例如，在一次运行中，一个任务可能在另外一个任务读变量 $y$ 之前读变量 $x$；在下一次运行中，事件的发生顺序可能与之相反。在大多数情况下，两个事件的执行顺序无关紧要。在其他情形中，顺序至关重要，为了确保程序的正确性，程序员必须引入同步来强制确保执行的顺利。为此目的而提供的原语将在本书的*实现机制*设计空间中进行探讨（见 6.3 节）。

**同步与异步**。我们使用这两个术语来定性地指明两个事件在时间上如何紧密耦合。如果两个事件必须在同一时间发生，则它们是同步的；否则，它们是异步的。例如，消息传递（即 UE 之间通过发送和接收消息而产生的通信），如果发送的消息必须等到接收者收到之后发送者才可以继续操作，则是同步的。如果发送者不管接收者而继续计算，或者接收者在等待接收完成的同时可以继续计算，则是异步的。

**竞态条件**。竞态条件是并行程序中特有的一种类型错误。当程序的结果随着 UE 的相对调度顺序的改变而发生变化时会出现该错误。由于操作系统而非程序员控制 UE 的调度，因此竞态条件导致了程序可能在同一个系统上使用同一份数据却产生不同的结果。竞态条件是
17 难以进行调试的错误，因为这类错误的结果具有不可重现性。程序可能已正确运行了 1000 次，却突然在第 10001 次运行时发生错误，当程序员想通过调试尝试重现错误时，运行结果却再次正确了。

竞态条件的错误归咎于同步。如果多个 UE 读写共享变量，程序员必须保护好对这些共享变量的访问，不管任务交叉执行的方式如何，以保持读写的顺序正确。当共享多个变量或者通过多层间接访问它们时，通过检查验证条件竞争不存在是非常困难的。可以通过工具帮助检测和修复竞态条件错误，例如，英特尔公司的 Thread Checker，这个问题仍然出现在活跃并且重要的研究领域 [NM92]。

**死锁**。死锁也是并行程序中一种特殊的错误。在某个任务周期中，每个任务因为等待另一个任务以便继续运行而阻塞时会发生死锁。因为所有的任务都在等待另外一个任务，所以它们会一直阻塞下去。举一个简单的示例，假定在消息传递环境中有两个任务，任务 $A$ 尝试从任务 $B$ 中接收数据，之后再给任务 $B$ 发送一条回复消息。与此同时，任务 $B$ 尝试从任务 $A$ 中接收数据，之后将发送消息给任务 $A$。由于每个任务都在等待另外一个任务首先给它发送消息，因此这两个任务会一直阻塞下去。幸运的是，发生死锁并非难事，因为任务会在死锁的那个点上停止。

## 2.5 并行计算的度量

实现并行程序的两个主要目的是获取更佳性能和解决更大问题。性能能够被建模和度量，因此本节将从另外一个视角审视并行计算，通过提供一些简单的分析模型来描述影响并行程序性能的某些因素。

考虑到计算由 3 部分组成：准备、计算和结束，因此程序在 PE 上运行的总时间由这 3 部分时间组成。

$$T_{\text{total}}(1) = T_{\text{setup}} + T_{\text{compute}} + T_{\text{finalization}} \tag{2-1}$$

当我们在拥有多个 PE 的并行计算机上运行这个计算时会发生什么情况？假定准备与结束部分不能和任何其他活动并发执行，计算部分被划分成运行在多个 PE 上的独立任务，而整个计算步骤的总数与最初的一样。当然可以通过式（2-2）描述在 PE 上的总计算时间，但是这是一个非常理想化的情形。尽管如此，很实际的想法是将计算划分成串行部分（这部分中额外的 PE 没有用武之地）和并行部分（这部分中更多数量的 PE 会减少运行时间）。因此，这个简单的模型包含了这个重要的关系。

$$T_{\text{total}}(P) = T_{\text{setup}} + \frac{T_{\text{compute}}(1)}{P} + T_{\text{finalization}} \tag{2-2}$$

18

度量使用额外的 PE 能带来多大的效用的一种重要方法是使用*相对加速比* $S$，通过它可以得知程序在并行情况下与在串行情况下的运行时间相差多少，从而描述问题运行到底有多快。

$$S(P) = \frac{T_{\text{total}}(1)}{T_{\text{total}}(P)} \tag{2-3}$$

与之相关的度量是效率 $E$，它是加速比与所使用 PE 数目的比值。

$$E(P) = \frac{S(P)}{P} \tag{2-4}$$

$$= \frac{T_{\text{total}}(1)}{PT_{\text{total}}(P)} \tag{2-5}$$

理想情形下，我们希望加速比的值为 $P$（PE 的数目）。这种情况通常称为*完全线性加速比*。遗憾的是，这是一个理想的情形，实际上很难实现，因为程序的准备和结束时间不能通过增加更多的 PE 而减少。程序中不能并发运行的部分称为串行部分，它们的运行时间占程序总时间的比率，称为*串行比率*，表示为 $\gamma$。

$$\gamma = \frac{T_{\text{setup}} + T_{\text{finalization}}}{T_{\text{total}}(1)} \tag{2-6}$$

于是，程序的并行部分占程序总时间的比率为 $1-\gamma$，因此，在拥有 $P$ 个 PE 的计算环境

下，整个计算时间的表达式可重写为式（2-7）。

$$T_{\text{total}}(P) = \gamma T_{\text{total}}(1) + \frac{(1-\gamma)T_{\text{total}}(1)}{P} \tag{2-7}$$

现在，借助新的表达式 $T_{\text{total}}(P)$ 对 $S$ 进行重写，得到了著名的 Amdahl 定律：

$$S(P) = \frac{T_{\text{total}}(1)}{\left(\gamma + \frac{1-\gamma}{P}\right)T_{\text{total}}(1)} \tag{2-8}$$

$$= \frac{1}{\gamma + \frac{1-\gamma}{P}} \tag{2-9}$$

可见，在并行部分没有开销的理想并行算法中，加速比的计算应该遵循式（2-9）。当使
19 用超多数量的处理器并且采用理想的并行算法时，到底能取得多大的加速比？假定表达式中的 $P$ 趋向于无穷大，得到 $S$ 的极限值如式（2-10）所示。

$$S = \frac{1}{\gamma} \tag{2-10}$$

因此式（2-10）给出了一个算法所取得的加速比的上限值，$\gamma$ 表示整个计算时间中串行部分所占的时间的比率。

这些概念对于并行算法的设计者而言至关重要。在设计一个并行算法时，理解算法中的串行比率非常重要，因为借助它可以得知算法所的性能。如果一个并行算法的串行部分高于10%（10% 比较常见），那么想把它实现成复杂并且可任意扩展的并行算法是行不通的。

当然，Amdahl 定律所基于的假设条件在实际中可能成立也有能不成立。在现实中，很多因素可能导致程序的运行时间长于公式所表示的时间。例如，创建额外的并行任务可能会增加开销和对共享资源访问的冲突几率。另一方面，如果当初的串行计算受限于资源而不是 CPU 的时钟周期，算法取得的实际性可能远高于使用 Amdahl 定律预测的结果。例如，在大型并行机上，内存能够容纳更大规模的问题，因此可以减少虚拟内存页，或者多个处理器，每个处理器拥有各自的缓存，可能允许更多的问题驻留在缓存之中。Amdahl 定律依赖的假设是：对于任意给定的输入，并行实现和串行实现执行计算的步骤数量是完全一样的。如果公式中使用的串行算法可能不是该问题的最佳算法，那么一个好的并行算法是通过采用不同的计算方式，使构建出的算法能够减少计算步骤的总数目。

在 [Gus88] 中可以看到，Amdahl 定律在某些情况（同一问题规模运行在数量变化的处理器上）是错误的。如果并行程序是气象模拟应用，则当新增处理器时，我们可能更倾向于在模型中添加更多细节，在问题规模增大的同时，保持运行时间不变。在这种情况下，利用 Amdahl 定律或者问题规模固定时的加速比来衡量并行程序的故率时，额外处理器所能带来的优势可能并不明显。

为说明这一点，我们重写加速比公式，得出了 $P$ 个处理器系统上能取得的加速比性能。在前面的式（2-2）中，我们得到 $T$ 个处理器的执行时间 $T_{\text{total}}(P)$ 是根据算法在单个处理器上串行部分和并行部分的执行时间推导出来的。在此，我们反行其道，根据串行和并行部分在 $P$ 个处理器上的执行时间推导出 $T_{\text{total}}(1)$。

$$T_{\text{total}}(1) = T_{\text{setup}} + PT_{\text{compute}}(P) + T_{\text{finalization}} \tag{2-11}$$

现在我们定义所谓的*可扩展的串行比率*，表示为 $\gamma_{\text{scaled}}$，如式（2-12）所示：

20

$$\gamma_{\text{scaled}} = \frac{T_{\text{setup}} + T_{\text{finalization}}}{T_{\text{total}}(P)} \tag{2-12}$$

从而有：

$$T_{\text{total}}(1) = \gamma_{\text{scaled}} T_{\text{total}}(P) + P(1 - \gamma_{\text{scaled}}) T_{\text{total}}(P) \tag{2-13}$$

通过对加速比公式（2-3）的重写和简化，我们得到了可扩展（或者固定时间）加速比[⊖]。

$$S(P) = P + (1 - P)\gamma_{\text{scaled}} \tag{2-14}$$

这个公式得到的加速比与 Amdahl 定律得到的结果一样，但是，当处理器数目增加时，可以考虑改变问题的规模。由于 $\gamma_{\text{scaled}}$ 依赖于 $P$，极限值不能直接获得，从而可以得到与 Amdahl 定律相同的结果。尽管如此，假如我们取 $P$ 的极限值，并且保持 $T_{\text{compute}}$ 不变，因此 $\gamma_{\text{scaled}}$ 的值是常量。这种情形可以解释为：当处理器数目增加时，通过增加问题规模来保持算法总的执行时间不变（这隐式地假定了伴随着问题规模的增长，串行部分的执行时间保持不变）。在这种情形下，加速比与 $P$ 呈线性关系。因此，当处理器数目较少，通过增加更多的处理器来解决某个固定的问题时，使用 Amdahl 定律来预测加速比的极限值比较合适。如果问题规模随着处理器数据的增加而增大，Amdahl 定律就不太适用了。文献 [SN90] 对于这两种加速比模型，以及内存固定的加速比版本进行了讨论。

## 2.6 通信

### 2.6.1 延迟和带宽

一个简单而实用的模型对消息传递的总时间的描述如下：一个固定开销加上一个依赖于消息长度的可变开销。

$$T_{\text{message-transfer}} = \alpha + \frac{N}{\beta} \tag{2-15}$$

这个固定开销 $\alpha$ 称为延迟，实际上它是指在通信介质上发送一条空消息所花费的时间，即从发送函数被调用到接收方完成数据接收的时间。延迟（以某种合适的时间单位给出）包括：软件和网络硬件开销、消息在通信介质中的传输时间。带宽 $\beta$（以某种合适的字节单位给出）是通信介质容量的度量。$N$ 表示消息的长度。

由于延迟和带宽都依赖于所使用的硬件，以及实现通信协议的软件质量，因此在系统之间差异较大。这些值能够通过一些简单的基准程序 [DD97] 测量，有时测量 $\alpha$ 和 $\beta$ 的值是非常有价值的，因为它们能够指导优化，用于改善通信性能。例如，在一个 $\alpha$ 值相对较大的系统中，尝试这样重构程序是值得的，即将程序存在的大量短消息聚集成少量长消息进行发送。 21
[BBC+03] 中给出了最近几个系统的数据。

### 2.6.2 重叠通信和计算以及延迟隐藏

如果我们进一步观察单个处理器上单个任务的计算时间，会发现它可以粗略地划分成计算时间、通信时间和闲置时间。通信时间指发送和接收消息所花的时间（因此仅适合于分布式内存机器），而闲置时间是指没有任务工作的时间，因为任务正在等待某个事件，例如，释放一个被其他任务所占用的资源。

---

⊖ 这个公式也称为 Gustafson 定律，由 E. Barsis 在 [Gus88] 中提出。

通常情况下，当任务等待系统中传播的某条消息时，可能会出现闲置。当发送一条消息（由于 UE 执行之前等待一条应答消息）或者接收一条消息时，都可能发生闲置。有时通过重构任务可以消除这种情形：通过发送消息和 / 或侦听（post）接收操作（也就是表示它想接收消息），然后继续计算。这使得程序员可以重叠计算和通信。图 2-7 展示了使用该技术的一个示例。这种消息传递方式对于程序员而言更加复杂，因为等待接收完成的任务是在与通信完全重叠之后，所以，程序员必须谨慎对待。

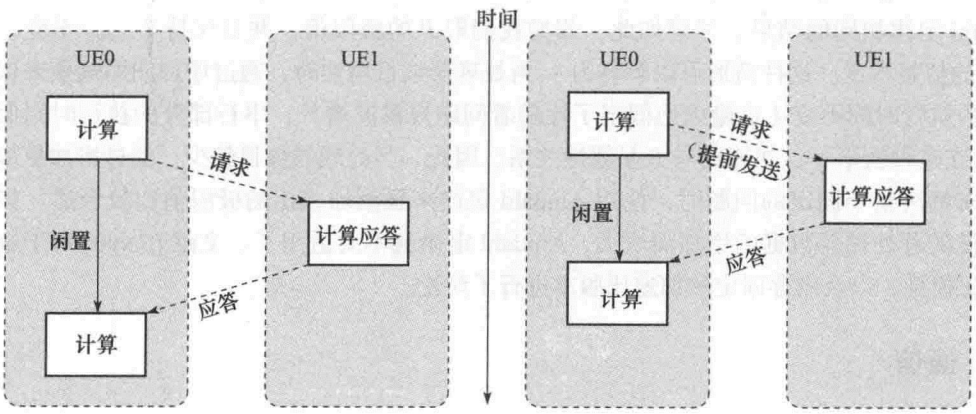



图 2-7 没有（左）和拥有（右）通信与计算重叠支持的通信。尽管在右边的计算中 UE 0 由于等待 UE 1 的应答，仍然存在一些闲置时间。由于 UE 1 启动更早，减少了闲置时间，并且计算所需要的总时间也更少

另外一种在许多并行计算机中使用的技术是给每个 PE 分配多个 U，以便当一个 UE 等待通信时，可以通过上下文切换到另外一个 UE 上使处理器保持繁忙。这也是延迟隐藏的一个示例，越来越多地应用于现代高性能计算机系统中，最著名的示例是来自于 Cray 公司的
22 MTA 系统 [ ACC+90 ]。

## 2.7 本章小结

本章简要介绍了并行计算中使用的一些概念和术语。术语的定义可参见术语表，同时也探讨了并行计算中主要使用的编程环境：OpenMP、MPI 和 Java。本书通篇将在示例中使用这三种编程环境。关于 OpenMP、MPI 和 Java 的更多细节，以及如何使用它们编写并行程序将
23 在附录中介绍。

# “寻找并发性”设计空间

## 3.1 关于设计空间

软件设计者从事多个领域的工作。设计的过程从问题领域着手，该领域中的设计元素直接与解决的问题相关（如流体流动、决策树、原子等）。软件是设计的最终目标，因此在某一点上，设计元素变成与程序相关的某些因素（如数据结构和软件模型）。我们将这称为*编程领域*。尽管人们通常尝试尽快进入编程领域，但是过早地走出问题领域的设计者可能会错过一些重要的设计选项。

在并行编程中尤其如此，并行程序借助于在不同的处理单元上处理问题中不同的部分，尝试使用更少的时间解决更大的问题。尽管如此，当问题包含可挖掘的并发性，换言之，多个活动或者任务能够同时执行时，它才可行。但是，当问题映射到编程领域时，将很难看到挖掘并发性的机会。

因此，程序员应该通过分析问题领域中暴露的可挖掘的并发性来开始设计并行方法。我们对设计空间的这种分析过程称为*寻找并发性*（Finding Concurrency）设计空间。设计空间中模式有助于识别和分析问题中可挖掘的并发性。这步完成之后，选择*算法结构*空间中一个或多个模式，用于设计合适的算法结构以挖掘已识别的并发性。

24

该设计空间及其所在模式语言中位置的总体概况如图 3-1 所示。

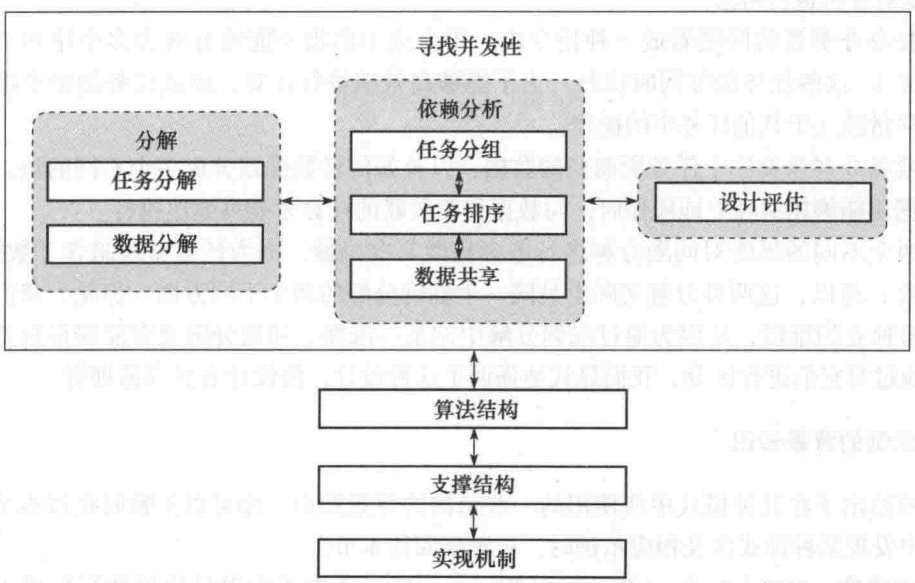



图 3-1　寻找并发性设计空间及其所在模式语言中位置的总体概况

在一个熟悉领域工作的有经验设计者，可以迅速发现可挖掘的并行性，并直接投入到算

法结构（Algorithm Structure）设计空间的模式中。

### 3.1.1 概述

在开始使用设计空间中的这些模式之前，算法设计者首先必须考虑所要解决的问题，并且明确是否有必要创建一个并行程序：问题规模是否足够大，结果是否意义重大，以至于必须采用并行方法来更快地解决问题？如果这样，那么下一步就是确保问题中的关键特征和数据元素被很好地诠释。最后，设计者需要理解问题中哪些部分是计算密集的，因为并行化任务的精力应当聚焦在这些部分。

在分析完成之后，就可以使用寻找并发性设计空间中的这些模式来开始设计一个并行算法了。设计空间中的这些模式可以分成3组。

- **分解模式**（decomposition pattern）：使用两种分解（任务分解和数据分解）模式可以将问题分解成多个能够并发执行的小片段。
- **依赖性分析模式**（dependency analysis pattern）：该组包含3种模式，用于任务分组和任务之间的依赖性分析；任务分组、任务排序和数据共享。通常这是模式的使用顺序。尽管如此，常常需要重新使用另一种模式，甚至可能需要重新使用分解模式。
- **设计评估模式**（design evaluation pattern）：这是设计空间中的最后一种模式，通过分析目前已经完成的工作，指引算法设计者转移到算法结构设计空间中的模式。这种模式是非常重要的，因为最佳的设计通常不是首次尝试就能发现的，越早识别出设计缺陷，就越容易纠正它们。通常情况下，设计空间中模式的使用是一个迭代过程。

25

### 3.1.2 使用分解模式

设计一个并行算法的第一步是将问题分解为多个能够并发执行的单元。我们可以从如下两个维度对分解进行考虑。

- 任务分解维将问题看成一种指令流，指令流中的指令能够分解为多个序列（称为任务），这些任务能够同时执行。为了能够高效地进行计算，组成任务的指令操作应当尽量独立于其他任务中的操作。
- 数据分解维关注于任务所需要的数据，以及如何将数据划分成多个不同的块。仅当数据块能够相对独立地操作时，与数据块相关联的计算才能高效地执行。

从两个不同的维度对问题分解进行考虑稍微有些机械。因为任务分解暗含了数据分解，反之亦然；所以，这两种分解实际上是同一个基础分解的两个不同方面。但是，我们仍将它们分成为独立的维度，是因为通过强调分解中的某一维度，问题分解通常能够很自然地进行下去。通过对它们进行区分，我们显式地强调了这种设计，使设计者更容易理解。

### 3.1.3 示例的背景知识

本节给出了在几种模式中所使用的一些示例的背景知识。你可以先暂时跳过本节，在后面阅读中发现某种模式涉及相应示例时，再重新阅读本节。

**医学成像**。PET（Positron Emission Tomography，正电子发射计算机断层扫描）提供了一个重要的诊断工具，使得内科医生能够观察到放射性物质是如何穿透病人身体的。遗憾的是，由放射射线的分布所组成的图像具有很低的分辨率，因为射线在穿透身体时会发散。根

据绝对射线强度进行推断也是比较困难的，因为穿透身体的不同路径对射线强度的削弱程度不同。 26

为了解决这个问题，使用射线如何在身体中进行传播的模型来纠正这个问题。通常的方法是构造一个 Monte Carlo 模型，Ljungberg 和 King [LK98] 对该模型进行了描述。在身体内随机选择某一点，假设在该点发射射线（通常是伽马射线），并跟踪射线的轨迹。当粒子（射线）穿过身体时，它会被所经过的不同器官削弱，直到离开身体，并被相机模型使用，从而定义了一条全轨迹。为了生成一个具有统计意义的模拟，需要跟踪成千上万条轨迹。

这个问题可以采取两种方式进行并行化。由于每条轨迹都是独立的，因此可以通过将每条轨迹与一个任务关联起来对这个应用进行并行化。这种方法将在 3.2 节中进行讨论。另一种方法是将身体划分为多个部分，然后将这些不同部分分配给不同的处理单元。该方法将在 3.3 节中进行讨论。

**线性代数**。线性代数在应用数学中是一个重要的工具：它提供了分析大型线性等式系统的解所需要的工具。典型的线性代数问题为：对于矩阵 $A$ 和向量 $b$，什么样的 $x$ 值满足下式：

$$A \bullet x = b \tag{3-1}$$

在线性代数中，式（3-1）中的矩阵 $A$ 占有中心地位。许多问题可以表示为这个矩阵的变换。这些变换依靠于矩阵乘法操作：

$$C = T \bullet A \tag{3-2}$$

如果 $T$、$A$ 和 $C$ 是秩为 $N$ 的方阵，则矩阵相乘得到矩阵 $C$ 中的每一个元素可以根据下式求得：

$$G_{i,j} = \sum_{k=0}^{N-1} T_{i,k} \cdot A_{k,j} \tag{3-3}$$

式中，下标表示矩阵的对应的元素。换句话说，矩阵 $C$ 第 $i$ 行和第 $j$ 列的元素是矩阵 $T$ 第 $i$ 行和矩阵 $A$ 第 $j$ 列元素的点积。因此，计算具有 $N^2$ 个元素的矩阵 $C$ 中每一个元素需要 $N$ 次乘法和 $N$–1 次加法，使得矩阵乘法的总体计算复杂度为 $O(N^3)$。

对矩阵乘法操作进行并行化的方法有很多。采用的并行策略要么使用基于任务的分解（正如 3.2 节所讨论的一样），也可以使用基于数据的分解（正如 3.3 节所讨论的一样）。

**分子动力学**。分子动力学用于对大型分子系统的运动进行仿真。例如，分子动力学仿真显示了大型蛋白质是如何移动的，以及不同形状的药物是如何与蛋白质相互作用的。因此， 27
分子动力学在配药行业中占据非常的重要地位并不奇怪。对于从事并行计算的计算机科学家，它也是一个非常有用的测试用例：简单明了，问题规模很大，很难高效地并行化。因此，它已经成为许多研究领域中的主题 [Mat94、PH95、Pli95]。

分子动力学的基本思想是将分子看作球的大型集合，球之间通过弹簧进行连接。球表示分子结构中的原子，而弹簧表示原子之间的化学键。分子动力学模拟是一个明确的时间步过程。在每个时间步中，计算出每个原子所受的作用力，然后使用标准的经典力学技术来计算作用力是如何在原子之间进行移动的。这个过程在每个时间步中重复进行，进而计算出分子系统的一条轨迹。

计算化学键（“弹簧”）所产生的作用力相对比较简单。这些力对应于化学键本身的振动和旋转。它们是短距离力，能够根据共享化学键的少数原子的信息来计算。主要的困难源自

于原子本身具有部分电荷。因此，当原子通过化学键仅与少数相邻原子进行相互作用时，电荷将使得每个原子对其他每个原子产生一个作用力。

这是著名的 $N$ 体（N-body）问题，需要计算大约 $N^2$ 项来寻找这些非化学键作用力。因为 $N$ 很大（几万或数十万），并且仿真中的时间步数目非常巨大（好几万），所以计算这些非化学键作用力所需要的时间占据了整个仿真中主要的计算时间。于是，人们提出了几种用于降低求解 $N$ 体问题所需要工作量的方法。在此仅讨论其中最简单的一种：*截止法*（cutoff method）。

该方法的思想非常简单。即使每个原子对其他所有原子都产生一个作用力，这个力也将随着原子之间距离的增大而减少。因此，可以确定一个距离，在该距离之外，由于力的作用很小，可以忽略。通过忽略超出这个截止点的原子，问题被归约为 $O(N*n)$，其中，$n$ 是截止空间范围内的原子数目，通常只有几百个。但计算量仍然巨大，它几乎占据了整个仿真时间，但至少问题已经解决了。

这个仿真存在很多细节，但基本过程如图 3-2 所示。

```
Int const N // number of atoms

Array of Real :: atoms (3,N) //3D coordinates
Array of Real :: velocities (3,N) //velocity vector
Array of Real :: forces (3,N) //force in each dimension
Array of List :: neighbors(N) //atoms in cutoff volume

loop over time steps
    vibrational_forces (N, atoms, forces)
    rotational_forces (N, atoms, forces)
    neighbor_list (N, atoms, neighbors)
    non_bonded_forces (N, atoms, neighbors, forces)
    update_atom_positions_and_velocities(
                    N, atoms, velocities, forces)
    physical_properties ( ... Lots of stuff ... )
end loop
```

图 3-2 分子动力学示例的伪代码

主要的数据结构包括：原子位置（atoms）、每个原子的速度（velocity）、作用在每个原子上的作用力（forces），以及每个原子截止距离范围内的原子列表（neighbors）。程序本事是一个时间步循环（time-stepping loop），循环中的每次迭代计算短距离力，更新邻域列表，然后寻找非化学键作用力。每个原子上的力计算出来后，借助一个简单的普通微分公式来计算原子的位置和速度。然后更新基于原子运动的物理属性，进入下一个时间步。

28 目前存在多种可以并行化分子动力学问题的方法。我们考虑最常用的方法，从任务分解（在 3.2 节中讨论）开始，紧接着进行相关数据分解（在 3.3 节中讨论）。这个示例展示了如何组合这两种分解模式来设计并行算法。

## 3.2 任务分解模式

1. 问题

如何将一个问题分解成多个可以并发执行的任务？

2. 背景

每一个并行算法设计的出发点都是相同的，即很好地理解待求解的问题。程序员必须理解问题的哪些部分是计算密集的部分，关键数据结构以及随着问题求解的展开如何使用数据。

下一步是定义组成问题的任务以及任务隐含的数据分解。基本上，每个并行算法都包含一个能够并发执行的任务集合。面临的主要挑战是挖掘这些任务和构建一个算法，使这些任务能够并发执行。

在某些情况下，问题能够很自然地分解为一个独立的（或接近独立）任务集合，并能够很容易开始一个基于任务的分解。在其他情况中，隔离任务可能非常困难，因此，更好的切入点是数据分解（将在 3.3 节对此进行讨论）。通常无法确定哪一种方法是最佳的，算法设计者经常需要同时考虑这两种方法。

尽管如此，无论切入点是基于任务的分解还是基于数据的分解，并行算法最终都需要多个可以并发执行的任务，因此，必须能够识别出这些任务。 29

3. 面临的问题

在这一点上，影响设计的主要因素是：灵活性、效率和简单性。

**灵活性**。设计中的灵活性将使得它能够适应不同的实现需要。例如，使设计局限于特定的计算机系统或者编程风格，通常不是一个明智的选择。

**效率**。仅当并行程序能够随着并行计算机的规模有效地扩展（减少运行时间或者内存使用量）时，它才是有用的。对于任务分解，这意味着我们需要足够多的任务让所有 PE 保持繁忙，每个任务具有足够的工作量，弥补用于管理任务之间依赖所带来的开销。但是，基于效率的驱动可能会导致缺乏灵活性的复杂分解。

**简单性**。任务分解需要足够复杂以完成工作，但也需要足够简单以便于程序调试和维护。

4. 解决方案

一个有效的任务分解的关键是确保任务之间充分地独立，以使得管理任务之间依赖所带来的开销仅占程序整个执行时间的一小部分。确保任务的执行能够均匀地分配在所有 PE 上也是非常重要的（负载平衡问题）。

在一个理想世界中，编译器可以为程序员挖掘任务。遗憾的是，这几乎从来没有出现过。取而代之的是，必须具备问题背景知识并熟悉实现的代码，然后手动完成。在某些情况下，可能需要完全重新将问题构造为另一种形式，以暴露相对独立的任务。

在基于任务的分解中，我们将问题看作由不同任务组成的一个集合，特别注意的是：

- 解决问题所采取的策略。（这些策略能够足以让目标机器上的处理单元保持繁忙吗？）
- 这些策略是截然不同且相对独立的吗？

在第一遍中，我们尽可能识别出尽可能多的任务。开始时具有很多任务并在随后合并它们，要比开始具有太少的任务并在随后拆分它们容易得多。

可以在许多不同的地方找到任务。

- 在某些情况下，每个任务对应于函数的一个不同调用。为每个函数调用定义一个任务有时候称为功能分解。 30
- 寻找任务的另一个地方位于算法内循环的不同迭代中。如果循环的每一层迭代是独立的，并且具有足够多的迭代，则基于任务的分解能够很好地起作用，将每一层迭代映

射到一个任务。这种基于任务分解的类型有时候称为循环分割（loop-splitting）算法。

- 任务在数据驱动的分解中也起着重要的作用。在这种情况下，大型数据结构将被分解，多个执行单元并发地更新数据结构的不同块。在这种情形下，任务是单个数据块上的更新工作。

另外，也需要注意在3.3节中给出的问题。

- **灵活性**。在产生任务数目方面，设计需要具备灵活性。通常通过适当的尺度参数化任务的数目和大小来完成，使设计能够适应多种拥有不同处理器数目的并行计算机。
- **效率**。在任务分解中，需要考虑两个主要的效率问题。首先，每个任务必须包括足够量的工作，以弥补创建任务和处理它们的依赖性所带来的开销。其次，任务的数目应当足够大，在整个计算期间，使得所有的执行单元都保持繁忙，并且做有用的工作。
- **简单性**。任务的定义方式便于调试和维护。如果可能，所定义的任务应当可以重用解决相关问题的现有串行程序。

识别出任务后，下一步就是查看任务中暗含的数据分解。*数据分解模式*将有助于这种分析。

5. 示例

**医学成像**。考虑3.1.3节描述的医学成像问题。在这个应用中，身体模型中某一点的选择是随机的，在该点上发生放射性衰变，并跟踪发射粒子的轨迹。为了创建一个具有统计意义的仿真，需要跟踪成千上万条轨迹。

将一个任务与每一条轨迹相关联是很自然的。并发管理这些任务非常简单，因为它们之间完全独立。另外，具有大量数目的轨迹，因此将具有很多任务，这使得分解能够适合于多种计算机系统，从具有较少处理单元的共享内存系统到具有数百个处理单元的大型集群。

基本任务定义完成后，现在我们考虑相应的数据分解，即定义与每个任务相关联的数据。
31 每个任务需要拥有定义轨迹的信息，但并不是全部：任务同样需要访问身体模型。尽管可能在问题描述中并不明显，但身体模型可能非常庞大。由于它是一个只读模型，因此在一台高效的共享内存系统中将不会产生问题；每个任务在需要时能够读取数据。但是，如果目标平台基于分布式内存的体系结构，身体模型需要复制到每个PE上。这将是非常耗时的，并且会耗费大量的内存。当系统中每个PE上具有较少的内存，并且PE之间网络的速度较慢时，基于身体模型的问题分解可能会更加高效。

这在并行编程中是常见的情形：许多问题主要基于数据或者主要基于任务进行分解。如果基于任务的分解避免了分解和分配复杂的数据结构，则编写程序和调试都将变得非常简单。另一方面，如果内存或网络带宽是一个限制因素，则集中于数据的分解可能会更加高效。一种分解方法比另一种方法“好”并不是什么问题，问题是需要平衡机器的需求和程序员的需要。3.3节将对此进行更加详细的讨论。

**矩阵乘法**。考虑两个矩阵相乘（$C=A \cdot B$），描述见3.1.3节。对于这个问题，我们可以产生一个基于任务的分解，分解方式是将最终结果矩阵的每个元素的计算作为一个独立的任务。每个任务需要访问$A$的一行和$B$的一列。这种分解方式的好处是所有的任务都是独立的。因为任务间共享的所有数据（$A$和$B$）是只读的，所以在一个共享内存环境中很容易实现。

但是这个算法的性能会很差，考虑到这3个矩阵都是秩为$N$的方阵。为了计算$C$中的

每一个元素，需要 ***A*** 中的 *N* 个元素和 ***B*** 中的 *N* 个元素，对于 *N* 次乘法 / 加法操作，将导致 2*N* 次内存访问。内存的访问速度比浮点算术运算慢，因此内存子系统的带宽将限制算法的性能。

较好的策略是设计一种这样的算法：能够最大化重用已加载到处理器缓存中的数据。我们可以采用两种不同方式来得到这种算法。首先，将前面定义的任务进行分组，使得使用矩阵 ***A*** 和矩阵 ***B*** 中的相似元素的任务在同一个 UE 上（见 3.4 节）。或者，可以从数据分解开始，设计算法的切入点就围绕着矩阵被装载到缓存中的方式进行。3.3 节将对这个示例进一步讨论。 32

**分子动力学**。考虑 3.1.3 节所描述的分子动力学问题。这个示例的伪代码又一次显示在图 3-3 中。

```
Int const N // number of atoms

Array of Real :: atoms (3,N) //3D coordinates
Array of Real :: velocities (3,N) //velocity vector
Array of Real :: forces (3,N) //force in each dimension
Array of List :: neighbors(N) //atoms in cutoff volume

loop over time steps
   vibrational_forces (N, atoms, forces)
   rotational_forces (N, atoms, forces)
   neighbor_list (N, atoms, neighbors)
   non_bonded_forces (N, atoms, neighbors, forces)
   update_atom_positions_and_velocities(
                N, atoms, velocities, forces)
   physical_properties ( ... Lots of stuff ... )
end loop
```

图 3-3 分子动力学示例的伪代码

在进行任务分解前，需要更好地理解问题的某些细节。首先，neighbor_list() 的计算是非常耗时的。计算的核心是对每个原子进行循环，在循环体的内部检查其他所有原子，以确定其他原子是否落入该原子的截止空间。幸运的是，时间步非常小，并且在任意给定的时间步中原子的移动距离很小。因此，这个耗时的计算仅仅在每 10 ~ 100 步进行一次。

其次，physical_properties() 函数计算能量、纠正系数和大量有意思的物理属性。尽管如此，这些计算是非常简单的，不会显著影响程序的整体运行时间，因此，我们将在本次讨论中忽略它们。

由于大多数计算时间位于 non_bonded_forces() 中，因此我们所选择的问题分解必须能够使计算高效地并行执行。在时间循环中的每个函数具有相似的结构（在串行版本中，每个函数包含一个作用于多个原子的循环操作，用于计算对作用力向量的贡献），依据这一个事实，问题分解将变得更加容易。这样，一个很直观的任务定义是：每个原子所需要的更新对应于串行版本中循环的一次迭代。完成任务分解之后，我们会获得如下任务：

- 寻找原子上振动作用力的任务；
- 寻找原子上旋转作用力的任务；
- 寻找原子上非化学键作用力的任务；
- 更新原子位置和速度的任务；

- 为所有原子更新邻居列表的任务（串行处理）。

通过手动收集任务，我们可以考虑其伴随的数据分解。主要的数据结构是邻居列表、原
33 子坐标、原子速度和作用力向量。每次更新作用力向量时，需要原子的邻居坐标。然而，计算非化学键作用力时可能需要所有原子的坐标，因为被仿真的分子可能以不可预计的方式折叠回来。我们将使用这些信息进行数据分解（在3.3节中）和数据共享分析（在3.6节中）。

**知名应用**。基于任务的分解在并行计算领域中极其常见。例如，距离几何代码DGEOM [Mat96]使用的是基于任务的分解，并行WESDYN分子动力学程序[MR95]也是如此。

## 3.3 数据分解模式

1. 问题

如何将一个问题的数据分解为多个相对独立操作的单元？

2. 背景

并行算法设计者必须详细了解所要求解的问题。此外，设计者应当识别出问题中计算密集的部分、求解问题所需要的关键数据结构，以及随着问题求解的展开如何使用数据。

对基本问题了解之后，并行算法设计者应当考虑组成问题的任务，以及任务中隐含的数据分解。为了创建一个并行算法，必须采用任务分解和数据分解。问题不是进行哪一种分解，而是先从哪一种分解开始。如果以下情况为真，基于数据的分解是个很好的切入点。

- 问题中计算密集的部分是围绕着一个大型数据结构的管理而组织的。
- 相似的操作应用于数据结构的不同部分，这样可以相对独立地操作数据结构中不同的部分。

例如，在许多线性代数问题中，需要更新大型矩阵，对矩阵的每一个元素使用相似的操作集。在这些情形中，通过观察矩阵是如何分解为多个并发更新的数据块，然后进行并行算法设计是比较容易的。任务的定义参考数据块的定义方式，并映射到并行计算机的处理单元上。

34 3. 面临的问题

在这里，面临的主要问题是灵活性、效率和简单性。

- **灵活性**。设计中的灵活性将使得它能够适应不同的实现需要。例如，将设计局限于特定的计算机系统或者编程风格，通常不是一个明智的选择。
- **效率**。仅当并行程序能够随着并行计算机的规模有效地扩展（减少运行时间或者内存使用量），它才是有用的。
- **简单性**。任务分解需要足够复杂以完成工作，但也需要足够以简单便于程序调试和维护。

4. 解决方案

在共享内存的编程环境（例如OpenMP）中，数据分解常常隐含于任务分解中。尽管如此，在大多数情况中，数据分解需要手动完成，因为内存是物理分布的，并且数据之间的相关性太复杂，不能明确对数据进行分解，或在NUMA架构的计算机上获得可以接受的效率。

如果基于任务的分解已经完成，则数据分解由每个任务的需要所驱动，如果每个任务能够与定义完善且不同的数据相关联，数据分解将变得容易。

尽管如此，当并行算法设计从数据分解开始时，我们不需要查看任务，但需要查看定义问题的主要数据结构，考虑它们是否能够分解为多个可以并发操作的数据块。包含以下几种常见的示例。

- **基于数组的计算**。可以根据更新数组中不同的数据块来定义并发性。如果数组是多维的，则可以采用多种方式对它进行划分（行、列或者不同形状的块）。
- **递归数据结构**。例如，我们可以考虑通过将一个大型的树数据结构分解为多个能够并发更新的子树来对这个数据结构进行分解。

不考虑底层数据结构的内在含义，如果数据分解是驱动问题求解的主要因素，则它将作为并行算法的组织原则。 35

当考虑如何分解问题的数据结构时，需要牢记竞争因素。

- **灵活性**。数据块的大小和数目应当灵活支持不同类型的并行系统。一种方法是使用一些参数来控制所定义的数据块的大小和数目。这些参数定义一些*粒度块*（granularity knob），通过改变它们可以修改数据块的大小，从而匹配底层硬件的需求。（但是注意的是：关于粒度，许多设计并不是无限制地适应。）

查看粒度对数据分解的影响最容易的地方是：管理数据块之间依赖所需要的开销。管理依赖性所需要的时间必须远小于总体运行时间。在一个好的数据分解中，相较于每个数据块所关联的计算量，依赖性的可扩展尺度很低。例如，在许多有限差分的程序中，数据块之间的边界单元（即数据块表面）必须是共享的。相关单元集的大小随着表面区域的大小而扩展，而计算所需的工作量随着数据块的容量而扩展。这意味着计算量可扩展（基于数据块的容量），以抵消数据依赖所带来的开销（基于数据块的表面区域）。

- **效率**。数据块必须足够大，以使得更新数据块的工作量能够抵消管理数据依赖所需要的开销。更加棘手的问题是考虑如何将数据块映射到 UE 上。一个高效的并行算法必须能够平衡 UE 间的负载。如果处理不当，某些 PE 分配的工作量可能不成比例，整体的可扩展性将受影响，这可能需要巧妙的方式来分解问题。例如，如果问题从左到右处理矩阵中的列，则矩阵的列映射将出现问题，这样会使得具有最左边列的 UE 优先于其他 UE 完成工作。基于行的数据块分解或者甚至是数据块周期分解（在该分解中，把行周期性地分配给 PE）都能够更好地让所有处理器保持繁忙。这些问题将在 5.10 节中进行更详细的讨论。
- **简单性**。过度复杂的数据分解使得调试非常困难。数据分解通常需要将一个全局索引空间映射到一个局部的任务索引空间。这种映射抽象使得数据分解很容易独立和测试。

数据分解之后，如果还没有进行这一步，紧接着是查看数据所隐含的任务分解。任务分解模式可能有助于这种分析。

5. 示例

**医学成像**。考虑 3.1.3 节所描述的医学成像问题。在这个应用中，身体模型中某一点的选择是随机的，在这一点上发生衰变放射性，并跟踪发射粒子的轨迹。为了创建一个具有统计意义的模拟，需要跟踪成千上万条轨迹。 36

在这个问题基于数据的分解中，身体模型是一个大型的核心数据结构，可以围绕着它组

织计算。身体模型被划分成多个片段，一个或多个片段与一个处理单元相关联。在轨道计算期间，这些身体片段是只读的并且不可写。因此，身体模型的划分不会导致数据之间的依赖。

在数据分解之后，我们需要查看与每个数据片段相关联的任务。在这种情形下，每条轨迹所需要的数据段定义成一个任务。轨迹在数据段内初始化并传播。当遇到数据段边界时，轨迹必须在段之间进行移动。这种移动定义了数据块之间的依赖性。

另一方面，在这个问题的基于任务的方法中（在3.2节中讨论），每个粒子的轨迹驱动算法设计。每一个PE可能需要访问整个身体模型以服务它的轨迹集合。在一个共享内存环境中这很容易，因为身体模型是一个只读数据集。但是，在一个分布式内存环境中，这将需要大量的启动开销，因为身体模型需要在系统中进行广播。

这是并行编程中较常见的情形：不同的观点将导致不同的算法出现，根据这些算法取得的性能差别很大。基于任务的算法比较简单，仅当每个处理单元访问一个大内存时，或者将数据加载到内存产生的开销相对于程序的运行时间微不足道时，它才起作用。另一方面，由数据分解所驱动的算法能够高效使用内存，并且（在分布式内存环境中）较少使用网络带宽，但在计算并发部分期间会产生很多通信开销，并且显得更加复杂。具体选择哪一种方法是非常困难的，在3.7节中将对此进一步讨论。

**矩阵乘法**。考虑两个矩阵的标准相乘（$C=A \cdot B$），如3.1.3节所描述的，可以采用几种基于数据的分解来解决这个问题。最直接的方法是将矩阵$C$划分为行数据块的集合（相邻行的集合）。根据矩阵乘法的定义，计算$C$中每行数据块的元素需要整个$A$矩阵，但仅需要矩阵$B$的对应行。根据这样的数据分解，算法的基本任务变成计算矩阵$C$中行数据块的元素。

更高效的方法是将3个矩阵都划分成多个子矩阵或数据块，这样不需要复制整个$A$矩阵。于是基本任务变成了$C$矩阵数据块的更新，随着计算的进行，$A$矩阵数据块和$B$矩阵数
37 据块周期性地出现在这些任务中。尽管如此，这种划分方法对于编程来说更加复杂，在问题的时间临界部分，必须认真协调通信和计算。4.6节和5.10节将进一步讨论这个示例。

矩阵乘法问题的特征之一是：浮点操作（$O(N^3)$）相对于内存访问（$O(N^2)$）的比率很小。这隐含着一个非常重要的方面，即考虑对内存的访问模式，以最大化重用缓存中数据。最高效的方法是使用数据块（子矩阵）划分，并调整数据块的大小，使得问题适合于缓存。通过仔细对任务进行分组，能够获得相同的算法，3.2节介绍过这种方法。但是从数据分解开始，将每个子矩阵的更新视为一个任务便于理解。

**分子动力学**。考虑3.1.3节和3.2节描述的分子动力学问题。很自然地采用任务分解来解决这个问题，其中任务是作用力计算例程中循环的某层迭代。

在此，概括问题及其任务分解，得到以下几种任务：

- 寻找原子上振动作用力的任务；
- 寻找原子上旋转作用力的任务；
- 寻找原子上非化学键作用力的任务；
- 更新原子任务和速度的任务；
- 为所有原子更新邻居列表的任务（串行处理）。

关键的数据结构包括：

- 原子坐标的数组，每个原子对应数组中一个元素；

- 原子速度的数组，每个原子对应数组中一个元素；
- 列表的数组，每个原子对应数组中一个元素，数组中每个元素定义了原子截止距离内所有原子的邻居；
- 原子上作用力的数组，每个原子占用数组中的一个元素。

速度数组的元素仅被拥有相应原子的任务所使用，这些数据之间不需要共享，因此，可以作为任务的局部数据。但是，每个任务需要访问整个坐标数组。因此，在一个分布式内存环境中，需要复制这些数据；而在一个共享内存环境中，需要在 UE 间共享它。

更有趣的是作用力数组。根据牛顿第三定律，原子 $i$ 对原子 $j$ 的作用力与原子 $j$ 对原子 $i$ 的作用力大小相同且方向相反。当我们可以根据这种对称性累加作用力时，可以将计算量减半。作用力数组中的值直到最后一步更新坐标和速度后才计算出来。因此，所使用的方法是：在每个 PE 中初始化整个作用力数组，并且将任务所累加的作用力项的部分和放置到这个数组之中。当所有的部分作用力项计算完成之后，将所有 PE 上的数组累加在一起获得最终的作用力数组。3.6 节将进一步讨论这个方法。 38

**知名应用**。数据分解方法在并行科学计算领域中非常普遍。并行线性代数库 ScaLAPACK [Sca、BCC+97] 使用了基于数据块的分解。用于稠密线性代数问题的 PLAPACK 环境 [vdG97] 使用了一种稍微不同的数据分解方法。例如，如果等式的形式为 $y=Ax$，并不是首先划分矩阵 $A$，而是以一种自然的方式划分向量 $y$ 和 $x$，然后确定对矩阵 $A$ 的划分。该论文的作者宣称利用这种方法取得了更好的性能，并且更容易实现。

在分子动力学示例中使用的数据分解由 Mattson 和 Ravishanker [MR95] 提出。关于这个问题更高级的数据分解方法由 Plimpton 和 Hendrickson 在文献 [PH95、Pli95] 中进行了讨论，该方法在节点数目众多的情况下具有良好的扩展性。

## 3.4 分组任务模式

1. 问题

如何将组成问题的任务进行分组，以简化任务之间相关依赖的管理工作？

2. 背景

使用了相应的任务和数据分解之后，可以应用这种模式。

这种模式描述了分析问题分解中任务之间依赖的第一步。在开发问题的任务分解时，我们考虑能够并发执行的任务。而在任务分解期间，我们并没有对它进行强调。很显然这些任务不能构成一个平滑的集合。例如，在算法中源于相同高级操作的任务可以很自然地分组在一起。其他的任务在源问题中可能并不相关，但是在它们并发执行中具有相似的约束，因此，这样也能够分组在一起。

总之，对于任务集合来说，具有大量的结构。这些结构——这些任务的分组——简化了问题之间的依赖性分析。如果一个分组共享一个时间约束（例如，在一个分组能够开始读取一个文件之前，等待另一个分组完成对这个文件的填写），则我们可以一次性对于整个分组满足这个约束。如果任务的一个分组必须工作在一起，对一份共享数据结构进行操作，则可以一次性对于整个分组设计所需要的同步。如果任务集合是独立的，则将它们合并到一个分组中，把它们作为单个大型的分组进行调度，这样可以简化设计，并增加可利用的并发性（从 39

而解决方案能够扩展到更多的 PE 上执行）。

每一种情况中，宗旨是定义共享约束的任务分组，并且通过处理分组而不是单个任务来简化管理约束的问题。

3. 解决方案

任务间的约束分为以下几个主要类别。

- 最容易理解的相关性是时间相关性——也就是关于任务集合执行顺序的一个约束。例如，如果任务 *A* 依赖于任务 *B* 的结果，则任务 *A* 必须等待，直到任务 *B* 执行完毕后才能继续执行。通常，我们从数据流方面考虑这种情形：任务 *A* 因等待任务 *B* 提供的数据而阻塞；当 *B* 完成时，数据流进入 *A*。在某些情形中，*A* 可以在数据刚从 *B* 处开始流入时就开始计算（例如 4.8 节描述的流水线算法）。
- 当一个任务集合必须同时运行时，出现另一种类型的顺序约束。例如，在许多数据并行问题中，初始问题领域被分解为多个能够并行更新的领域。任何给定区域的更新通常需要它邻域的边界信息，如果所有的区域不能够同时处理，并行程序可能停顿，或者发生死锁，因为某些区域需要等待其他非活跃区域的数据。
- 在某些情形下，分组中的任务相互之间是完全独立的，这些任务之间的执行不具有顺利约束。这是任务集合的一个重要特征，因为这意味着它们能够以任意顺序进行执行，包括并发执行，清楚这一点非常重要。

这种模式的目标是基于这些约束对任务进行分组，原因如下。

- 通过分组任务，简化了任务间部分顺序的构建，因为顺序约束能够应用于组，而不是单个任务。
- 分组任务可以更容易识别出哪些任务必须并发执行。

对于一个给定的问题和分解，可能存在多种分组任务的方式。我们的目标是寻找一种能够简化相关性分析的分组方式。为了阐明这一点，将相关性分析作为发掘和满足程序并发执
40 行的约束。当任务共享一个约束集时，将它们分组可以简化相关性分析。

不存在寻找任务分组的单个方式。我们建议的方法如下，要牢记，不能只考虑任务分组，而忽略约束本身。在设计的这个时刻上，最好做到尽可能抽象——识别约束并且分组任务有助于解决它们，但尽量不要拘泥于细节。

- 首先，查看初始问题是如何分解的。在大多数情形中，一个高级操作（例如，求解某个矩阵）或一个大型迭代的程序结构（例如，某个循环）在定义分解中扮演着关键的角色。这是寻找任务分组的首选。高级操作相应的任务可以很自然地分组在一起。

  在此刻，任务可能存在很多小的任务分组。在下一步中，我们将查看分组中任务间共享的约束。如果多个任务共享一个约束——通常是共享数据结构的更新——将它们视为另一个分组保存。算法设计需要确保这些任务同时执行。例如，许多问题包含对某个任务集中共享的数据结构进行协作更新。如果这些任务不能并发执行，则程序可能产生死锁。
- 紧接着，查看其他任何任务分组是否共享相同的约束。如果共享，则合并这些分组。大型任务分组提供了额外的并发性，让更多的 PE 保持繁忙，也为任务执行的调度提供了额外的灵活性。因此，使得平衡 PE 间的负载更加容易（也就是说，确保了每个 PE 在问题求解上耗费近似相同的时间量）。

- 下一步是观察任务分组间的约束。当分组具有明确的时间顺序时，或者当清晰的数据链在分组间移动时，这是非常容易做到的。但是，更加复杂的情形是当其他独立任务分组共享分组间的约束时。在这些情形中，将这些分组合并为一个更大的独立任务分组是非常有意义的——因为大型任务分组通常使得调度更加灵活并且扩展性更好。

4. 示例

**分子动力学**。该问题的描述见 3.1.3 节，3.2 节和 3.3 节讨论了它的分解，识别出的任务如下：

- 寻找原子上振动作用力的任务；
- 寻找原子上旋转作用力的任务；
- 寻找原子上非化学键作用力的任务；
- 更新原子位置和速度的任务；
- 为所有原子更新邻居列表的任务（单个任务，因为这部分计算是串行的）。

41

考虑如何将这些任务分组在一起。在第一遍中，前面列表中的每一项对应于原始问题中的某个高级操作，并且定义一个任务分组。但是，如果进一步挖掘这个问题时，将看到隐含在作用力函数中的更新是相互独立的。唯一的相关性是对作用力进行累加，并存放到一个作用力数组中。

接着查看是否可以合并这些分组。通过观察列表，会发现前面两个分组中的任务是相互独立的，但共享相同的约束。在这两个分组中，坐标对于原子的小部分邻居来说是只读的，并且得出作用力数组的局部分布，因此我们能够将它们合并为一个分组，用于求取化学键的相互作用力。其他分组由于具有不同的时间或顺序约束，因此不能合并。

**矩阵乘法**。3.2 节讨论了将矩阵乘法 ***C=AB*** 分解为多个任务，每个任务对应于矩阵 ***C*** 中某个元素的更新。但是，最新计算机的内存组织擅长于粗粒度任务，例如，更新矩阵 ***C*** 的一个数据块，正如 3.3 节描述的一样。精确地讲，这等价于将元素更新任务进行分组，每个任务对应于一个数据块。采用这种方法进行分组能够充分利用系统内存。

## 3.5 排序任务模式

1. 问题

给定一种将问题分解为任务的方式，以及一种将这些任务收集到逻辑相关分组中的方式，如何对这些任务分组进行排序，以满足任务间的约束？

2. 背景

这种模式给出了分析某个问题中任务间依赖性的第二步。其中第一步（在 3.4 节已讨论）是基于任务间的约束分组任务。这里讨论的下一步是根据任务集合执行顺序上的约束来寻找并正确阐述依赖性。任务间的约束主要分为如下几类。

- 时间依赖性，即关于任务集合执行顺序的约束。
- 一些特定的任务必须同时执行的需求（例如，每个任务所需要的信息将由其他任务产生）。

42

- 缺乏约束，即相互间是完全独立的。但是严格地说，这并不是一种约束，但它是任务

集的一种重要特征，因为这意味着这些任务可以按任意顺序执行，包括并发执行，了解这一点是非常重要的。

- 这种模式有助于根据一个任务集合执行顺序上的约束来发掘并且正确解释其依赖性。

3. 解决方案

当识别任务之间的顺序约束和定义任务分组间的部分顺序时，有两个目标需要满足。

- 顺序必须是充分严格的，以满足所有的约束，确保最终的设计是正确的。
- 对顺序的限制不应该比它需要的严格。过度约束的解决方案限制了设计选项，并且可能降低程序的效率；约束越少，就能够更加灵活地转换任务，便于平衡 PE 间的计算负载。

为了识别顺序约束，考虑如下几种任务依赖方式。

- 首先，在执行一组任务之前，查看它们所需要的数据。当识别这些数据之后，寻找创建这些数据的任务分组，并且将出现一种顺序约束。例如，如果一组任务（称为 *A*）构建了一个复杂的数据结构，另一组任务（称为 *B*）使用了该数据结构，则这两个分组间存在一个串行的顺序约束。当这两个分组被组合到一个程序中时，它们必须串行执行：首先执行 *A*，然后执行 *B*。
- 另外考虑外部设备是否可以施加一些顺序约束。例如，如果一个程序必须以某种特定顺序写一个文件，那么这些文件 I/O 操作很可能施加一种顺序约束。
- 最后，非常重要的是，注意任何顺序约束不存在的情况，如果一些任务分组能够相互独立地执行，则存在很大的机会来挖掘并行性，因此我们需要注意任务是独立的情况，就像关注任务间是相互依赖的情况。

不管约束源头怎样，我们必须定义约束来限制执行顺序，以确保它们在算法设计中被正确地处理。同时，非常重要的是，注意缺少顺序约束的情况，因为这将对后面的设计带来非
43 常有价值的灵活性。

4. 示例

**分子动力学**。这个问题的描述见 3.1.3 节，3.2 节和 3.3 节讨论了它的分解。3.4 节描述了如何借助如下分组来组织这个问题的任务。

- 寻找每个原子上“化学键作用力”（振动作用力和旋转作用力）的任务分组。
- 寻找每个原子上非化学键作用力的任务分组。
- 更新每个原子位置和速度的任务分组。
- 为所有原子更新邻居列表的任务（通常会构成一个任务分组）。

现在我们准备考虑分组间的顺序约束。很显然，原子位置的更新直到作用力计算完成后才能够发生。此外，非化学键作用力直到邻居列表更新后才能够计算。因此，在每个时间步中，分组必须按照图 3-4 所示的顺序进行排列。

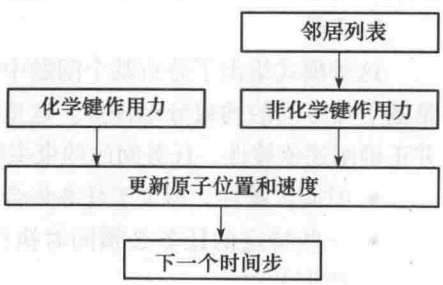



图 3-4 分子动力学问题中任务的顺序

目前，详细考虑如何强制执行这些顺序约束还有点为时过早，但最终我们需要提供某种类型的同步机制来确保严格遵守它们之间的顺序。

## 3.6 数据共享模式

1. 问题

对于给定问题的数据分解和任务分解，如何在任务间共享数据？

2. 背景

在高层，每一个并行算法包括：

- 一个能够并发执行的任务集合（参考 3.2 节）；
- 并发任务集合相应的数据分解（参考 3.3 节）；
- 管理任务间的依赖性以确保任务安全地并发执行。

44

正如 3.4 节和 3.5 节描述的，依赖性分析的切入点是基于任务间的约束来分组任务，然后确定任务分组所需要的顺序约束。紧接着，分析数据如何在任务分组间共享（在这里讨论），这样能够正确管理任务对共享数据的访问。

尽管分析会产生任务分组和任务间的顺序约束，但是分析主要集中于任务分解。在依赖性分析这个阶段中，注意力转移到数据分解，即将问题的数据分解为多个独立更新的数据块，每个数据块关联一个或者多个负责该数据块更新的任务。这种数据块通常称为“任务局部数据”（或者仅称为“局部数据”），因为它与负责更新它的任务是一种紧耦合的关系。然而，每个任务仅能够使用它自己的局部数据进行操作是非常罕见的；数据可能需要在任务间以多种方式进行共享。最常见的两种情况如下。

- 除了任务局部数据之外，问题的数据分解可能定义一些必须在任务间共享的数据：比如，一些任务可能需要协作更新某个大型的共享数据结构。这样的数据不能被任何给定的任务所标识；对于问题来说，它们是全局的。这些共享数据被多个任务修改，因此，它们是任务间依赖性的源头。
- 当一个任务需要访问另一个任务的局部数据时，数据依赖性也可能出现。这种类型的数据依赖性的经典示例存在于有限差分方法中，该方法使用数据分解来进行并行化，其中，问题空间中的每一个点需要使用附近点的值进行更新，因此对一个分解的数据块进行更新时需要它的邻居数据块的边界值。

这种模式讨论了并行算法中的数据共享，以及如何处理典型的共享数据。

3. 面临的问题

这种模式的目标是识别出哪些数据在任务分组间共享，并且确定如何正确而高效地管理共享数据的访问。

数据共享对正确性和效率具有重大影响。

- 如果共享处理不正确，任务可能因为某种竞态条件（race condition）而获得无效数据；这通常发生在共享地址空间的环境中。在该环境中，在任务读取内存位置中的某些数据时，这些数据还未写入。

45

- 为了保证共享数据已经准备且以待使用，可能导致过度的同步开销。例如，可以在读取共享数据之前放置一个栅栏操作[⊖]来加强顺序约束。但是，这样可能产生无法接受的

⊖ 栅栏是一种同步构造，是程序中定义的一个同步点，即一组 UE 必须都到达这一点之后，它们才能够继续执行下去。

低效率，特别是当仅有少数 UE 实际上共享数据的情况时。较好的策略是将数据复制到局部数据中，或者重新构建任务，以最小化对共享数据的读取次数。

- 另一种数据共享的开销源是通信。在一些并行系统中，对共享数据的访问暗含了在 UE 间的消息传递。这个问题时常可以通过重叠通信和计算来缓解，但这并不总是可行的。通常，较好的选择是精心构造算法和任务，使得任务间共享数据所需的通信量最小。另一种方式是在每个 UE 拥有共享数据的一份副本，尽管需要确保所复制的数据在 UE 中保持一致，但这种方式比较高率。

因此，其目标是管理共享数据以确保其正确性，并且尽可能不对效率造成太大的影响。

4. 解决方案

首先是识别出任务间的共享数据。

当分解主要是基于数据的分解时，共享数据很明显。例如，在一个有限差分问题中，把基本数据分解成多个数据块。分解的本质表明：数据块的边缘数据被相邻数据块所共享。本质上，当基本分解完成后，数据共享就已经实现了。

当基于任务的分解占据主导地位时，情况会更加复杂。在任务定义的某时刻，确定数据是如何被传入或传出任务的，以及数据是否被任务所更新。这些是潜在的数据共享源。

当识别出共享数据后，需要分析如何使用这些数据。共享数据主要分成如下 3 类。

- **只读**。数据可以读，但不能够写。因为它是不可修改的，所以访问这些值不需要任何的保护措施。在某些分布式内存系统中，对只读数据的复制是值得的，这样每个执行单元自身拥有共享数据的一份副本。
- **有效 – 局部**（effectively-local）。数据被分解为多个子集，每个子集仅被一个任务访问（读或写）。（例如，数组被多个任务共享，共享的方式是数组中的元素实际上分解为多个任务局部的数据集合）。这种情况为处理依赖性分析提供了某些选择。如果能够独立访问数据的子集（如数组元素，不一定是列表元素），则不需要担心对数据访问的保护。在分布式内存系统中，这样的数据通常分布于各个 UE 中，每个 UE 仅拥有它的任务所需要的数据。在计算结束时，如果需要，可以将数据重新合并成单个数据结构。

46

- **读 – 写**。数据既可以读，也可以写，并且被多个任务所访问。这是非常常见的情形，包括各种复杂情形，在该情形中，数据被任意数目的任务所读或写。这种情形是最难处理的，因为对数据的任何访问（读或写）必须利用某种类型的互斥访问机制（锁、信号量等）来保护，这具有非常昂贵的代价。

值得提及两种常见的特殊读写情形。

- **累加**。所用的数据被累加到一起而获得一个结果（例如，当归约计算时）。对于共享数据中的每一个位置，值由多个任务进行更新，其中更新发生在某种类型的结合累加操作中。最常见的累加操作包括：求和、求最小值和求最大值，可能是对一些操作数对的任意结合操作。对于这样的数据，每个任务（或者通常是每个 UE）拥有一份独立的副本；累加发生在这些局部副本中，当局部累加结束完成后，所有累加结果再被累加到单个全局副本中。
- **多次读 / 单次写**。数据被多个任务读取（所有任务都需要它的初始值），但仅被单个任务所修改（该任务能够以任意频率对它进行读和写）。这些变量经常出现在基于数据

分解的算法中。对于这种类型数据，至少需要两个副本，一个用于保存初始值，另一个被修改它的任务使用；当不再需要时，可以丢弃包含初始值的副本。在分布式内存系统中，通常需要为访问（读或写）数据的每个任务创建一个数据副本。

5. 示例

**分子动力学**。对这个问题的描述见 3.1.3 节，这在 3.2 节和 3.3 节中进行了讨论。然后，识别出这些任务分组（在 3.2 节中），并考虑了任务分组间的时间约束（在 3.5 节中）。现在我们先将暂时忽略时间约束，只关注问题的最终任务分组中的数据共享，这些任务分组包括：

- 寻找每个原子上“化学键作用力”（振动作用力和旋转作用力）的任务分组；
- 寻找每个原子上非化学键作用力的任务分组；
- 更新每个原子位置和速度的任务分组；
- 为所有原子更新邻居列表的任务（轻松地构成了一个任务分组）。

47

这个问题中的数据共享非常复杂。图 3-5 中总结了分组间的共享数据。主要的共享数据项如下。

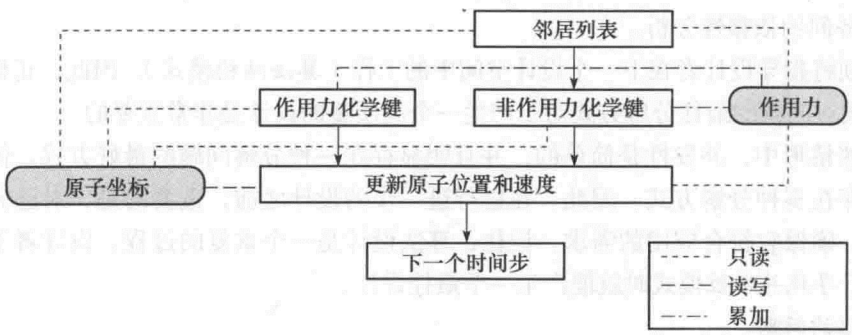



图 3-5 分子动力学中的数据共享（区分读、读写以及累加共享）

- 原子坐标，被每个分组使用。
- 这些坐标被化学键作用力组、非化学键作用力组和邻居列表更新组视为只读数据。这些数据对于位置更新组来说是可读写的。幸运的是，位置更新组在其他 3 个组完成后才能执行（基于使用 3.6 节开发的顺序约束）。因此，在前面 3 个分组中，我们可以对访问位置中的数据不加保护，或者复制它。对于位置更新组，位置数据属于读写类型，需要仔细控制对这些数据的访问。
- 作用力数组，除了邻居列表更新组之外，其他每个分组都使用。

这个数组对于位置更新组是只读数据，而被化学键作用力组和非化学键作用力组用作累加数据。因为位置更新组必须在作用力计算完成之后（由使用的排序任务模式所确定的），所以对于作用力分组将这个数组视为累加类型，对于位置更新组将它视为只读类型。

分子动力学仿真 [MR95] 的标准程序首先初始化作用力数组，作为每个 UE 上的局部数组。然后，每个 UE 计算对作用力数组元素的贡献，其中对精确项的计算是不可预知的，因为分子在空间上出现折叠。所有作用力计算完成之后，局部数组被归约到单个数组中，在每个 UE 上放置一份它的副本（更多信息请参考 6.4.2 节对归约的讨论）。

- 邻居列表，被非化学键作用力组和邻居列表更新组所共享。

48

对于邻居列表更新组来说，邻居列表本质上是局部数据；对于非化学键作用力计算来说，

邻居列表是只读数据。列表可以放在每个 UE 中的局部内存中进行管理。

## 3.7 设计评估模式

1. 问题

目前，分解和依赖性分析是否足够好，以至于可以转向于下一个设计空间，还是应当重新对问题进行设计？

2. 背景

此刻，问题已经被划分成多个能够并发执行的任务（使用任务分解模式和数据分解模式），并且任务之间的依赖性已经被识别出来（使用分组任务模式、排序任务模式和数据共享模式）。尤其是，原始问题经过分解和分析，产生的结果如下：

- 任务分解识别出多个能够并发执行的任务；
- 数据分解识别出从属于每个任务的局部数据；
- 分组任务和排序分组满足时间约束；
- 任务间的依赖性分析。

这四项将指导设计者在下一个设计空间中的工作（算法结构模式）。因此，正确使用这些项并且寻找对问题的最佳分解方式对于产生一个高质量的设计是非常重要的。

在某些情形中，并发性是简单的，并且明显存在一种分解问题的最好方式。但是，一个问题通常存在多种分解方式。因此，在进行进一步的设计之前，重要的是，对已完成的设计进行评估，确保它符合应用的需求。记住，算法设计是一个重复的过程，设计者不要期望在第一次执行寻找并发性模式时就能产生一个最佳设计。

3. 面临的问题

需要从 3 个方面来评估设计。

- **对目标平台的适用性**。类似处理器数目以及如何共享数据结构等问题将对任何设计的效率产生影响。然而，设计越依赖于目标体系结构，灵活性将会越差。
- **设计质量**。所期望的简单性、灵活性和效率可能会产生冲突。
- **为设计的下一阶段做准备**。任务和依赖性是规则的还是非规则的（即它们的大小是相似的，还是差异很大）？任务间的交互是同步的还是异步的（即交互是有规律的周期发生，还是差异很大甚至随机发生）？这些任务是否以一种高效的方式聚合在一起？明白这些问题将有助于从算法结构设计空间的模式中选择一种合适的解决方案。

49

4. 解决方案

在转向设计空间的下一个阶段之前，根据本节前面提到的 3 种角度去评估目前为止所做的工作，是有帮助的。这种模式的剩余部分由问题和讨论组成，有助于评估。

**对目标平台的适用性**。尽管设计者期望尽可能延迟将一个程序映射到一个特定的目标平台上，但是确实需要至少在评估一个设计时考虑目标平台的特征。下面是一些与目标平台的选择相关的问题。

**多少个 PE 可用**？在某些例外情形下，任务数多于 PE 数目时很容易使 PE 保持繁忙。很显然，在任务数目固定的情形下，我们无法充分利用更多的 PE，但如果每个 PE 只处理一个或少数几个任务时，负载均衡性会很差。例如，考虑蒙特卡罗仿真中的情形，在该仿真中，

计算针对随机选择的不同数据集不断重复执行，这样当数据差别很大时，计算所花费的时间也迥然不同。开发一个并行算法最容易想到的方法是将每个计算（针对某个独立的数据集）视为一个任务；这些任务是完全独立的，可以对它们随意调度。但由于每个任务所执行的时间差异很大，除非任务数目远超过 PE 的数目，否则很难获得较好的负载均衡。

这个规则的例外是：设计中任务的数目可以随着 PE 的数目进行调整，这样能够维护良好的负载平衡。此类设计的一个示例是 3.3 节中描述的基于数据块的矩阵相乘算法。在该算法中，任务对应于数据块，所有任务包含大致相等的计算量，因此调整任务的数目与 PE 的数目相等，将得到一个具有良好负载均衡的算法（但注意的是，即使在这种情形中，任务数目 50
超过 PE 数目也是很有好处的。例如，这将允许计算和通信重叠）。

**数据结构是如何在 PE 间共享的**？如果设计中包含任务间大规模或细粒度数据共享，如果所有的任务访问同一内存，这个设计将更加容易实现，并且效率更高。实现的难易程度取决于编程环境；基于共享内存模型（所有 UE 共享同一块地址空间）的环境使实现具有大量数据共享的设计更加容易实现。效率还依赖于目标机器，拥有大量数据共享的设计在一个对称多处理器机器上（其中所有的处理器对内存的访问时间是一致的）运行可能比在一台共享内存而物理内存是分布的机器上运行的效率高。相反，如果要在一个分布式内存体系结构上使用消息传递机制，则拥有大量数据共享的设计可能不是一个好的选择。

例如，考虑 3.2 节描述的医学成像问题中基于任务的方法。这个设计需要所有的任务读取一个非常大型的数据结构（身体模型）。在共享内存环境中，这不会出现问题；而在一个分布式内存环境中，如果每个 PE 自身拥有一个很大的内存子系统，并且具有足够高的网络带宽，可以处理对大型数据集的广播，这样也不会出现问题。但是，在一个内存和网路带宽资源有限的分布式内存环境中，需要强调数据分解在算法设计中的重要性，因为这能够提高内存的利用率。

需要细粒度数据共享（相同的数据结构被多个任务重复地访问，特别是包含读和写的操作时）的设计在一台共享内存机器中也可能运行的效率更高，因为对每个访问进行保护所带来的开销很可能小于在一台分布式内存机器中所需要的开销。

这些规则的例外是这样的一个问题：将任务间的大规模或细粒度共享的数据仅分配给相同的执行单位。这种方式很容易分组和调度任务。

**目标体系结构是如何隐含 UE 的数目，以及数据结构是如何在 UE 间之间共享的**？事实上，我们回顾前面的这两个问题，只考虑了 UE，而不是 PE。

重要的区别是：如果目标计算机系统依赖于在每个 PE 上使用多个 UE 以隐藏延迟。当考虑是否在每个 PE 上使用多个 UE 进行设计时，需要牢记两个因素。

首个因素是，目标计算机系统是否为每个 PE 上的多个 UE 提供高效的支持。某些系统提供这样的支持，比如，Cray MTA 机器和具有超线程功能的 Intel 处理器架构的机器。这样的架构对快速的上下文切换提供了硬件支持，使得它广泛应用于隐藏延迟的情形中。其他系统不能为单个 PE 上使用多个 UE 提供好的支持。例如，MPP 系统具有慢速的上下文切换，或 51
者 / 并且每个节点具有一个处理器，仅当每个 PE 只有一个 UE 时才能更好地运行。

第二个考虑的因素是，当每个 PE 上拥有多个 UE 时，设计是否能够很好地利用它们，例如，如果设计包含高延迟的通信操作，则为每个 PE 分配多个 UE，使得当某些 UE 在等待某个高延迟的操作时，其他 UE 可以继续运行，从而达到隐藏延迟的目的。但是，如果设计中

包含的通信操作是紧密同步的（例如，数据块发送/接收对）并且相对高效，则将多个UE分配给每个PE可能会妨碍设计的实现（因为需要投入额外的精力来避免出现死锁），而不是提高效率。

**在目标平台上，任务中做有意义的工作所耗费的时间是否比处理任务依赖性所耗费的时间多？**确定一个设计是否高效的关键因素是完成计算所耗费的时间与通信或同步所耗费的时间的比值。比值越高，程序的效率也就越高。这个比值不仅受设计所需协调事件的数目和种类的影响，还受目标平台特征的影响。例如，一个消息传递设计在一台具有快速互连网络和相对较慢的处理器的MPP机器上运行时，效率是可以接受的。但在一个由以太网互连的强大的工作站网络上运行时，效率可能变得更加低，可能无法接受。

值得注意的是，这个重要的比值同时也受问题规模的影响，而问题规模与可用的PE的数目息息相关。对于规模固定的问题，每个处理器完成计算所耗费的时间伴随着处理器数目的增加而减少，与此同时，每个处理器耗费在通信和同步的时间可能保持不变，或者甚至随着处理器数目的增加而增加。

**设计质量。**牢记目标平台的这些特征，我们可以从这三个方面来评估设计：灵活性、效率和简单性。

**灵活性。**我们希望高级设计能够适用大量不同的实现需求，当然是所有重要的需求。本节余下的部分将提供影响灵活性的部分因素的列表。

- 分解产生的任务数目是否灵活？这种灵活性使得设计能够适用大量不同类型的并行计算机。
- 任务分解所隐含的任务的定义是否独立于它们执行时的调度方式？这样的独立性使得负载平衡问题更容易得到解决。
- 数据分解中数据块的大小和数目是否参数化了？这样参数化使得设计更容易扩展到不同数目的PE。

52

- 算法的边界情况是否已经处理？一个好的设计将会处理所有相关的情形，甚至包括不常见的情形。例如，一个常用的操作是矩阵转置，使得矩阵的列数据块分布变为矩阵的行数块分布。对于矩阵的秩能被PE数目整除的方阵来说，很容易设计算法并编写代码。但是如果矩阵不是方阵，或者如果行数目大于列数目，并且行数目和列数目都不能被PE数目整除，又会怎样呢？这需要显著地改变转置算法。比如，对于一个长方形矩阵来说，存储数据块的缓冲区需要足够大，以至于能够容纳比两个数据块更大的容量。如果矩阵的行数目和列数目都不能被PE数目整除，则分配给每个PE的数据块的大小将不同。算法能够处理由于每个PE上数据块大小不同而造成的不均匀的负载吗？

**效率。**程序应当高效地利用可用的计算资源。本节剩余的部分将给出需要检查的部分重要因素的列表。注意的是，同时优化所有这些因素通常是不太可能的，因此在设计上的折衷是不可避免的。

- 计算负载能够是否在PE之间均匀分配？如果任务是独立的，或者如果它们的大小粗略相等，这个目标很容易达到。
- 开销是否已经最小化了？开销源于以下几种：UE的创建和调度以及通信和同步。UE的创建和调度存在开销，因此每个UE需要处理足够多的工作来抵消这些开销。另一

方面，UE 越多负载平衡越好。

- 通信也是一种主要的开销源，特别是依赖于消息传递的分布式内存平台。2.6 节已经讨论过，传输一条消息所使用的时间由两部分组成：源于操作系统开销和消息在网络上启动开销所产生的延迟开销，以及随着消息长度的增加所带来的开销。为了最小化延迟开销，应当最小化所发送的消息数目。换句话来说，小数目的大消息优于大数目的小消息。另外，通信也与网络带宽有关，这些开销有时候可以通过重叠通信和计算来隐藏。
- 在共享内存机器中，同步是一种主要的系统开销来源。当数据被 UE 共享时，为了避免竞争条件，某个任务需要等待另一个任务，这样会出现依赖。相比在 UE 上进行的大量操作，用于控制这种等待的同步机制产生的开销是非常昂贵的。此外，某些同步构造产生了显著的内存流量，因为它们占满了闪存、缓冲区和其他系统资源，以确保所有 UE 看到内存中的数据是一致的。这种额外的内存流量能够妨碍计算中显式的数 53
  据移动。通过保持数据对任务的局部化可以减少同步开销，从而最小化同步操作的频率。

**简单性**。爱因斯坦的名言：使事物尽可能简单，而不只是稍微简单一点。

需要时刻牢记：实际上所有的程序最终都需要调试和维护，并且常常需要增强功能和移植。一个设计——即使一个完美的设计——如果它太难于调试和维护，以及难以验证最终程序的正确性，这样的设计都是没有价值的。

3.1.3 节描述的医学成像示例在 3.2 节和 3.3 节进一步讨论，在支持简单性方面是一个很好的例子。在这个问题中，一个大型的数据块被分解，但是这种分解将迫使并行算法添加一些复杂的操作，用于在 UE 间传递轨道和分配数据库块。这种复杂性使得最终程序非常难以理解，并且使调试变得非常复杂。另一种方法是复制数据库，会得到一个非常简单并且拥有完全独立的任务的并行程序，这些任务可以分配给多个工作者。这样所有复杂的通信都消失了，程序的并行部分也非常易于调试和分析。

**为下一个阶段做准备**。借助*寻找并发性*模式进行问题分解时定义了一些重要的组件，这些组件将指导*算法结构*设计空间中的设计：

- 任务分解，识别能够并发执行任务；
- 数据分解，识别出从属于每个任务的局部数据；
- 分组任务和排序分组的方法以满足时间约束；
- 任务间的依赖性分析。

在转向下一个阶段之前，针对下面的一些问题来考虑这些组件。

**任务和它们的数据依赖性的规则度如何**？规则的任务是那些大小和工作量都相似的任务。不规则的任务是那些相互之间差别很大的任务。如果任务是不规则的，则任务的调度和它们的数据共享都将非常复杂，都需要在设计中注意。在一个规则的分解中，所有的任务在某种程度上都是相似的——大体相同的计算量（针对不同的数据集），在与其他任务的数据共享方面具有近似相等的依赖性。示例包括在 3.2 节、3.3 节和其他节中描述的各种矩阵相乘算法。 54

在一个不规则的分解中，每个任务所做的工作与数据依赖性在不同的任务中差异很大。例如，考虑一个大型系统中的某个离散事件仿真，该系统由大量不同的组件构成。我们可以为这个仿真定义一个并行算法，其方式是为每一个组件定义一个任务，并且让它们基于仿真

的离散事件进行交互。这将是一个非常不规则的设计，因为在已经完成的工作和与其他任务的依赖性方面，任务间存在巨大的差异。

**任务（或任务分组）间的交互是同步的还是异步的**？在某些设计中，任务间的交互在时间方面也是非常规则的——即同步的。比如，一个线性代数问题的并行化中包含对某个大型矩阵的更新，典型做法是在将矩阵划分给多个任务，每一个任务使用它自己的数据，利用矩阵的其他部分数据来更新自身拥有的矩阵部分。假设所有需要更新的数据在开始计算时提供，这些任务一般首先交换信息，然后独立地计算。另一种类型的示例是*流水线计算*（详情参考4.8节），其中我们对一个输入数据集序列执行一个多步操作，通过建立一个任务流水线（每个任务对应一个操作），当某个任务完成它的工作时，数据从一个任务流向另一个任务。当所有任务的步调基本一致时，这种方法将能够很好地工作——即如果它们的交互是同步的。

而在另外一些设计中，任务间的交互并不是那么规则。如前面描述的离散事件模拟示例，其中事件使得任务间的交互变得不规整。

**任务是以最佳的方式分组的吗**？时间关联非常容易识别：能够在同一时间内运行的任务能够很自然地分组在一起。但是一个高效的设计也需要基于整个问题中的逻辑关系对任务进行分组。

作为分组任务的一个示例，考虑3.4节、3.5节和3.6节的示例部分中所讨论的分子动力学问题。最终我们所得到的分组具有层次性（见3.4节）：基于问题的高级操作的相关任务分组，基于能够并发执行的那些任务进一步分组。这种方法更容易推导出设计是否满足所要求的约束（因为约束可以根据高级操作所定义的任务分组来描述），同时也考虑到了调度的灵活性。

## 3.8 本章小结

借助于*寻找并发性设计空间*中的模式揭示了问题中的并发性。分析过程中的关键元素如下所示：

- 任务分解，识别出多个能够并发执行的任务；
- 数据分解，识别出从属于每个任务的局部数据；
- 分组任务和排序分组的方式以满足时间约束；
- 任务之间的依赖性分析。

一门模式语言过去被描述成一张模式网，其中一种模式以逻辑方式连接到下一个模式，但是，*寻找并发性设计空间*的输出并不符合这种情况。而设计空间的目标是帮助设计者创建一些设计元素，这些元素一起通向模式语言的剩余部分。

第 4 章
Patterns for Parallel Programming

# “算法结构”设计空间

## 4.1 引言

设计并行算法的第一阶段是分析问题并明确可利用的并发性，这经常通过*寻找并发性*设计空间模式来实现。*寻找并发性*设计空间的输出是将问题分解为多个设计元素：

- 任务分解，识别出多个可以并发执行的任务；
- 数据分解，识别出每个任务所需要的局部数据；
- 任务分组和分组排序的方式，以满足时间约束；
- 任务间依赖性分析。

这些元素提供了从*寻找并发性*设计空间到*算法结构*设计空间的连接。*算法结构*设计空间的目标是细化设计，使得该设计更接近于在一台并行计算机上通过映射并发任务到多个 UE，实现并发执行的程序。

在定义算法结构的很多方式中，多数都遵循 6 种基本设计模式中的一种。这些模式组成了*算法结构*设计空间。这种设计空间的总体框图及其在模式语言中的位置如图 4-1 所示。

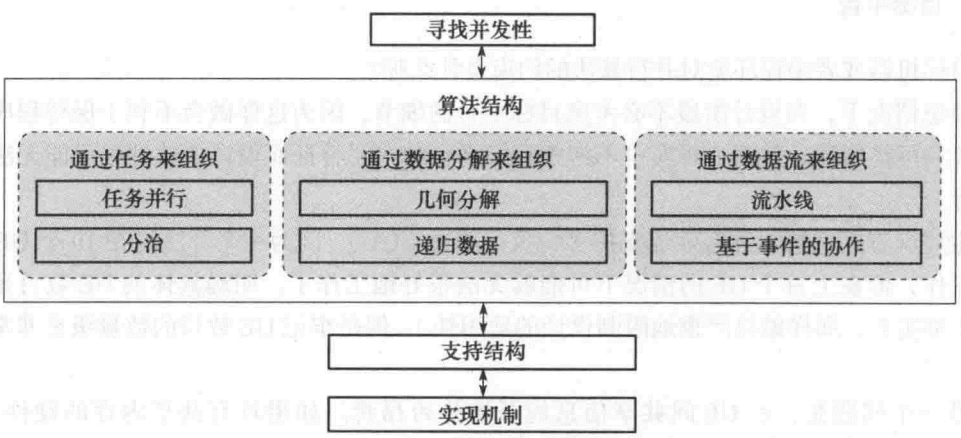



图 4-1 *算法结构*设计空间的总体框图及其在模式语言中的位置

该阶段的关键问题是确定哪种模式或者哪些模式最适合具体的问题。 57

首先，我们需要记住的是：从不同的方面分析问题，可以将设计引向不同的方向；从某一方面分析可能建议使用某种结构，而从另一个方面分析可能建议使用另外一种结构。但是，在几乎所有情况下，都需要牢记以下几点。

- **效率**。并行程序能够快速运行并充分利用计算机资源。
- **简单性**。一个简单的算法将产生易于理解的代码，且代码易于开发、调试、验证和修改。

- **可移植性**。理论上，程序应当能够在多种不同类型的并行计算机上运行。这将使某种特定程序的“市场”最大化。更重要的是，一个程序可以使用多年，而任何特定的计算机系统仅能够使用几年。可移植的程序保护了软件的投资。
- **可扩展性**。理论上，一个算法应当能够在从几个到数百甚至上千个处理元素（PE）上高效地运行。

但是，这些要求在几个方面都相互冲突。

效率与可移植性冲突：为了使得程序高效，几乎总要考虑运行程序的目标系统的特点，这样会限制程序的可移植性。结合特定系统或者编程环境特性的设计可以产生在特定系统上高效运行的程序，但是在其他不同的平台上变得不可使用，要么是因为性能太差，要么是在新的平台上难以实现或者根本无法实现。

58 效率与简单性冲突：例如，为了编写一个使用了*任务并行*模式的高效程序，有时需要采用复杂的调度算法。但是，在许多情况下，这些调度算法使程序变得难以理解。

因此，一个好的算法设计必须在以下两方面取得平衡：①抽象性和可移植性；②适合特定的目标体系结构。设计者面临的挑战，特别是在算法设计的初级阶段，是使得并行算法设计足够抽象来支持可移植性，但同时要确保它最终能够高效地在即将运行的并行系统上实现。

## 4.2 选择一种算法结构设计模式

通过考虑下面描述的问题，可以为给定问题寻找一种高效的*算法结构*模式。

### 4.2.1 目标平台

目标机器或者编程环境对并行算法的约束是什么呢?

理想情况下，在设计阶段不必考虑目标平台的细节，因为这样做会不利于保持程序的可移植性和可扩展性。然而，现实中不考虑目标平台的主要特征而设计的软件很可能无法有效地运行。

主要问题是系统能够有效地支持多少执行单元（UE），因为一个算法能在10个UE上很好地工作，但在上百个UE的情况下可能就无法很好地工作了。明确具体的UE数目是没必要的（事实上，那样做将严重地限制设计的适用性），但是牢记UE数目的数量级是非常重要的。

另一个问题是，在UE间共享信息的开销是否昂贵。如果具有共享内存的硬件支持，信息交换通过对存储器的共享访问实现，频繁的数据共享是可行的。然而，如果目标平台是通过慢速网络连接的节点集合，共享信息所需要的通信开销是昂贵的，必须尽可能避免。

考虑UE的数目和共享信息的开销这两个问题时，应避免过分地约束设计。通常，软件比硬件使用时间长，因此在软件的整个生命过程中，它可能被用在许多不同类型的目标平台上。设计的目标是获得一个能够在初始目标平台上工作良好的设计，但同时具有足够的灵活性，以适应不同类型的硬件。

最后，除了考虑多个UE和在UE间共享信息的方式外，并行计算机还有一个或者多个用于实现并行算法的编程环境。不同的编程环境提供不同的方式来创建任务和在不同UE之

间共享信息，所以，无法很好地映射到目标编程环境的特征的设计很难实现。 59

### 4.2.2 主要组织原则

考虑问题的并发性时，是否存在一种特殊方法用于查看是否支持并提供组织该并发性的高级机制？

寻找并发性设计空间中的模式分析，根据任务和任务分组、数据（共享的和局部任务的）、任务分组间的顺序约束描述了潜在的并发性。下一步是寻找一个算法结构来描述如何将这种并发性映射到 UE 上。通常并发性隐含了一些主要组织原则。这些组织原则主要分为三种类型：基于任务的组织、基于数据分解的组织和基于数据流的组织。现在我们进一步考虑这些组织原则的细节。

对于某些问题，在某一时刻，实际上只有一个任务分组处于活跃状态，这个任务分组内任务之间的交互方式是并发性的主要特征。所谓的易并行程序就是一个很好的例子，在该程序中，任务之间是完全独立的。还有一些程序，单个分组内的任务通过相互合作来完成一个结果的计算。

对于另外一些问题，以数据分解的方式和任务间数据共享的方式作为主要的方法来组织并发性。例如，许多问题关注少数大型数据结构的更新，考虑并发性最有效的方法是看这个结构是如何在 UE 之间分解和分配的。求解微分方程和进行线性代数计算的程序通常属于这一类，因为它们会频繁地对大型数据结构进行更新。

最后，对于某些问题，并发性的最大特征存在于明确定义的交互的任务分组中，关键问题是了解数据在任务间如何流动。例如，在信号处理应用中，数据从一个被组织为流水线的任务序列中流过，每个任务对连续的数据元素完成一次转换。或者，在一个离散事件的模拟中，可以通过将其分解为多个基于“事件”交互的任务，将其并行化。因此，并发性的主要特征是这些不同任务分组的交互方式。

此外，还需要注意的是，最有效的并行算法设计需要利用多个算法结构（以分层、合成和顺序方式组合），这也是考虑一个设计是否有意义的关键。例如，设计的最高水平常常是一个或者多个算法结构模式的顺序组合。而某些设计可能以分层方式组织，其中一个模式用于组织主要任务分组之间的交互，其他的模式用于组织各个分组内部的任务，例如，在流水线模式的实例中，其中每一个阶段都是任务并行模式的一个实例。

### 4.2.3 算法结构决策树

对于每一个任务子集，哪种算法结构设计模式能够最有效地完成从任务到 UE 的映射？ 60

考虑完前面几小节提出的问题后，通过很好地理解目标平台所强制的约束、分层和合成的角色，以及问题的一个重要组织原则，我们准备选择一个算法结构。决策由图 4-2 所示的决策树引导。从树的顶层开始，考虑并发性和主要的组织原则，使用这些信息选择树的 3 个分支之一，然后讨论所选的子树。注意，对于某些问题，最终的设计可能组合了多个算法结构：如果没有单个结构适合所解决的问题，则可能需要将组成问题的任务划分为两个或者多个分组，对于每个分组独立地进行这个过程，然后确定如何组合最终的算法结构。

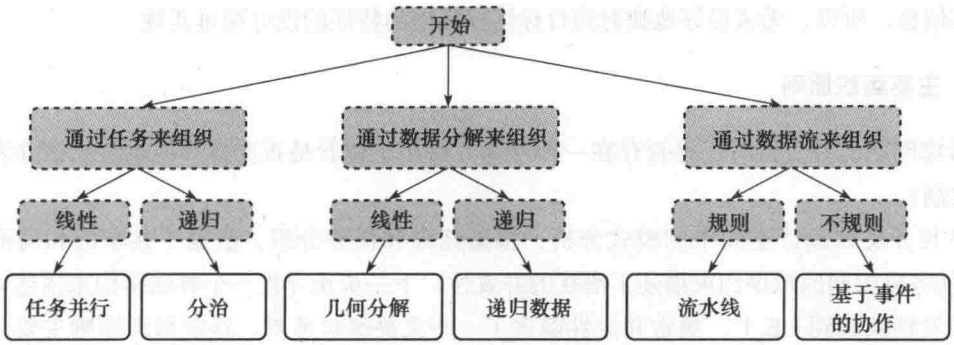



图 4-2 算法结构设计空间的决策树

**通过任务来组织。**当任务自身的执行是最好的组织原则时，选择通过任务来组织分支。然后确定任务是如何枚举的。如果这些任务可以聚集为一个任意维度的线性集合，选择任务并行模式。这种模式包含两种情形：一种是任务间相互独立（即所谓的易并行算法）；另一种是任务间有某些依赖，具体表现形式是任务间需要访问共享数据或者需要交换消息。如果任务由一个递归过程枚举，选择分治模式。在这种模式中，问题的求解方式是：将问题递归地划分为多个子问题，独立地求解每一个子问题，最后将这些子问题解决方案重新合并为原始问题的解决方法。

**通过数据分解来组织。**理解并发性过程中，当数据分解是主要的组织原则时，选择通过数据分解来组织分支。在该组织中存在两种模式，两种方式的不同点是分解的构造方式不同，一种是线性方式，另一种是递归方式。当问题空间被划分为多个离散的子空间，问题的求解通过计算每一个子空间的解决方案实现，而每一个子空间的解决方案通常需要使用其他几个
61 子空间的数据时，选择几何分解模式。在科学计算领域，这种模式非常常见，它在基于网格的并行计算中作用明显。当问题根据一个递归数据结构（例如，二叉树）定义时，选择递归数据模式。

**通过数据流来组织。**当主要的组织原则是数据流对任务分组强制一种顺序的方式时，选择通过数据流来组织分支。这种模式有两种，当顺序是规则的和静态的时，应用其中一种；当它是不规则的和动态的时，应用另外一种。当任务分组间的数据流是规则的、单向的，并且在算法期间（即任务分组可以组织为一个流水线，数据流流过该流水线）不发生变化时，选择流水线模式。当数据流是非规则的、动态的和/或不可预测（即通过事件同步，任务分组可以交互）时，选择基于事件的协作模式。

### 4.2.4 重新评估

算法结构模式适用于目标平台吗？对所选择的决策频繁地复审，以确信所选择的模式能够很好地适用于目标平台是非常重要的。

在设计中，选择一个或者多个算法结构模式之后，浏览它们的描述，以确信它们适合于目标平台（例如，如果目标平台由大量的通过慢速网络连接的工作站组成，并且所选择的算法结构模式之一需要频繁地进行任务间通信，可能很难有效地实现这个设计）。如果所选择的模式不能够适合目标平台，尝试另外一种组织原则，并重新进行前面的过程。

## 4.3 示例

### 4.3.1 医学成像

例如，考虑 3.1.3 节描述的医学成像问题。这个应用程序模拟了大量的伽马射线穿透身体，并输出到照相机的过程。描述并发性的一种方式是将每条射线的仿真定义为一个任务。因为它们在逻辑上是等价的，所以我们将它们放置到单个任务分组中。任务间仅有的共享数据是表示身体的一个大型数据结构，因为要访问的数据结构是只读的，所以任务间彼此独立。

因为该问题中具有许多独立的任务，所以可以较少地考虑目标平台的特征：任务数量较大，意味着我们能够高效地使用任意合理数目的 UE；任务之间相互独立，意味着任务间共享信息的开销不会对性能产生很大的影响。

这样，利用图 4-2 所示的决策树，我们应该能够选择一种合适的结构。在这个问题中，任务是相互独立的，我们所顾虑的问题是当我们选择一种算法结构后，如何将这些任务映射 62
到 UE。针对该问题，主要的组织原则是任务的组织方式，因此，我们选择从*通过任务来组织*分支开始。

现在我们考虑所指定的任务集的属性——任务集中的任务是按层次排列还是驻留在一个非结构化或平面的集合中。对于这个问题，任务位于一个非结构化的集合中，它们之间没有明显的层次结构，因此我们选择*任务并行*模式。注意，在这个问题中，我们可以利用任务是独立的这个事实来简化求解过程。

最后，我们从可能使用的目标平台重新评估该决策。正如我们前面所观察到的，这个问题的关键特征（任务数目巨大并且任务是相互独立的）使得我们不需要重新考虑这个决策，因为我们所选择的结构将难以在目标平台上实现。

### 4.3.2 分子动力学

以 3.1.3 节描述的分子动力学问题作为第二个示例。在*任务分解*模式中，与这个问题相关的任务分组如下：

- 寻找一个原子上振动作用力的任务；
- 寻找一个原子上旋转作用力的任务；
- 寻找一个原子上非化学键作用力的任务；
- 更新一个原子位置和速度的任务；
- 为所有原子更新邻居列表的任务。

在每一个分组内部，把任务表示为关于分子系统内部所有原子的每一次循环的迭代。

利用图 4-2 所示的决策树我们可以选择一个合适的算法结构。一种选择是根据任务分组间的数据流来组织并行算法。注意，仅有前面三个任务分组（振动作用力、旋转作用力和非化学键作用力的计算）可以并发地执行，即在更新原子位置、速度和邻居列表之前，必须完成对它们的计算。这不是完全的并发执行，因此应当使用图 4-2 中的不同分支来求解这个问题。

另一种选择是根据每一个分组中的任务集合来获得可挖掘的并发性，在这个示例中，任务是关于原子循环的每一次迭代。建议采用一种由线性排列的任务组织的方式，或者基于图

4-2 使用任务并行模式。可挖掘的总的并发性是非常大的（根据原子数目的数量级），这为设计并行算法提供了很大的灵活性。

对于这个问题，目标机器对并行算法具有重要的影响。数据分解模式中讨论的依赖性
63 （将坐标复制到每一个 UE 上，并对每一个 UE 上的部分和归约，以计算出一个全局作用力数组）表明将需要在 UE 间传递 $2\times3\times N$（$N$ 是所有原子的数目）项。但是，计算的数量级是 $nN$，其中 $n$ 是每一个原子的邻居原子的数目，而且远小于 $N$。因此，通信和计算具有相同的数量级，通信开销的管理将是设计算法时要考虑的关键因素。

## 4.4 任务并行模式

1. 问题

当把问题恰当地分解为一个能够并发执行的任务集合时，如何高效地挖掘这种并发性？

2. 背景

从根本上来说，每一个并行算法都是一个并发任务集合。这些任务及其之间的依赖性可以通过观察（对于简单的问题）或通过应用寻找并发性设计空间中的模式来识别。对于某些问题，聚焦于这些任务本身及其之间的交互可能不是组织算法的最佳方式，在某些情形中，根据数据（如在几何分解模式中）或者根据并发任务间的数据流（如在流水线模式中）组织任务是可行的。但是，在许多情形中，最好直接利用任务本身来组织。当直接基于任务设计算法时，称算法为任务并行算法。

任务并行算法分类非常多，包括以下一些示例。

- 任务分解模式中描述的医学成像示例中的射线追踪代码：与每一条“射线”相关的计算成为一个单独的并且完全独立的任务。
- 任务分解模式中描述的分子动力学示例：每一个原子上非化学键作用力的更新是一个任务。通过将作用力数组复制到每一个 UE 中，以获取每个原子作用力的部分和来处理任务间的依赖性。当所有的任务计算完它们对非化学键作用力的贡献后，每个作用力数组被组合（或“归约”）为一个存储每个原子的非化学键作用力总和的数组。
- 分支界限计算法：在该方法中，问题的求解方式是重复地从解决方案空间的一个列表中取出一个解决方案空间，检测它，或者声明它为一个解决方案并丢弃它，或者将其划分为多个更小的解决方案空间，然后将这些小的解决方案空间添加到解决方案空间列表中。通过将每一个“检测与处理一个解决方案空间”步骤视作一个单独的任务，
64 可以将计算并行化。在共享的任务队列中，任务间的依赖性逐渐减弱。

通常，问题可以分解为能够并发执行的任务集合。任务可以是完全独立的（如医学成像示例）或者任务间存在依赖性（如分子动力学示例）。在大多数情形中，任务将与循环的迭代相关，但也可以将它们与大规模的程序结构联系起来。

在许多情形中，所有的任务在计算开始时就已经知道（前面两个示例）。但是，在某些情形中，随着计算的展开，动态地创建任务，如分支界限示例。

另外，通常所有的任务在问题得到解决之前必须完成，但是对于某些问题，可能并不需要完成所有的任务，就可以解决问题。例如，在分支界限示例中，我们有一个对应解决方案空间的任务池以备搜索，在这个池中的所有任务完成前，我们可能已经找到一个可接受的解

决方案。

3. 面临的问题

- 为了挖掘问题中潜在的并发性，我们必须将任务分配给 UE。理想情况下，我们想以一种简单、可移植、可扩展和高效的方式完成这个工作。但是，如 4.1 节所述，这些目标可能是相互冲突的。要考虑的重点是如何平衡负载，即确保所有的 UE 具有大致相等的工作量。
- 如果任务以某种方式（通过顺序约束或者数据依赖性）相互依赖，则这些依赖性必须正确地处理。再一次记住以下几个有时会冲突的目标：简单性、可移植性、可扩展性和效率。

4. 解决方案

任务并行算法的设计包含 3 个主要元素：任务和它们的定义方式，任务间的依赖性，以及调度（任务怎样分配到 UE）。我们将对它们分别讨论，但事实上它们紧密相连，在确定最终决策前，必须全面地考虑三者。考虑完这些因素之后，我们需要再全面检查整个程序结构，然后检查这种模式的某些重要的特例。

**任务。**理想情况下，任务分解应当满足两个标准：第一，任务的数目至少与 UE 的数目一样多，当然越多越好，确保调度时具有很大的灵活性。第二，与各个任务相关联的计算量必须足够多，以此来抵消与任务管理和处理任何依赖性相关的开销。如果初始的分解不能满足这两个标准，则需要考虑是否采用另外一种能够满足这两个标准的分解方式。

例如，在图像处理应用中，每一个像素的更新是独立的，任务的定义可以是单独的像素、线条，甚至是图像中的一块。在一个具有少量节点并且由慢速网络连接的系统中，任务粒度 65
应当足够大，以弥补较大的通信延迟，因此基于图像块的任务是合理的选择。但是，在一个由快速（低延迟）网络连接的大量节点集合系统中，将需要较小的任务，以确保存在足够的工作来保持所有 UE 繁忙。注意，这里强调了对快速网络的需求，否则每个任务较小的工作量将不够补偿通信带来的开销。

**依赖性。**任务间的依赖性对算法设计具有重要的影响。存在两种类型的依赖性：顺序约束和与共享数据相关的依赖性。

针对该模式，顺序约束应用于任务分组，可以通过强制任务分组按照所需要的顺序来处理分组。例如，在一个任务并行的多维快速傅里叶变换中，需要变换的每一维都存在一个对应的任务分组，需要使用同步或其他的程序结构，确保在计算下一个维度之前，必须完成当前维度的计算。另外，我们可以将这个问题简化为任务并行计算的一个顺序组合，每一个步骤对应于一个任务分组。

共享数据依赖性可能更加复杂。最简单的情形是任务间没有依赖。令人惊奇的是，大多数问题都属于这种情形。这类问题通常称为易并行问题。它们是并行计算问题中最简单的并行程序，主要考虑的是任务如何定义（如前面所述）和调度（将在后面讨论）。当任务间共享数据时，虽然仍然存在一些相对容易处理的常见情况，但算法将变得更加复杂。我们可以将依赖性分成如下几类。

- **可消除的依赖性。**在这种情形中，任务间的依赖性并不是真正的依赖性，而是一种表面上的依赖性，可以通过简单的代码替换消除它。最简单的一种情况是临时变量，变量的作用域对于每一个任务来说都是完全局部的，即每一个任务初始化变量，而不用

引用考其他任务。这种情况可以通过为每一个 UE 简单地创建局部变量的一个副本来进行处理。在更复杂的情形中，循环迭代中的表达式可能需要转换为封闭形式的表达
66 式，以消除循环依赖性。例如，考虑下面的简单循环：

```
int ii = 0, jj = 0;

for(int i = 0; i< N; i++)
{
   ii = ii + 1;
   d[ii] = big_time_consuming_work(ii);
   jj = jj + i;
   a[jj] = other_big_calc(jj);
}
```

变量 ii 和 jj 引入了任务间的一个依赖性，阻止了循环的并行化。我们可以通过使用封闭形式的表达式替换掉 ii 和 jj，来消除这种依赖性（注意，ii 和 i 的值是相同的，jj 的值是从 0 到 i 的值的总和）：

```
for(int i = 0; i< N; i++){

   d[i] = big_time_consuming_work(i);
   a[(i*i+i)/2] = other_big_calc((i*i+i)/2));
}
```

- **“可分离的”依赖性**。当依赖性包含一个对共享数据结构的累计过程时，可以通过在计算开始时复制该数据结构，执行该任务，任务完成后合并所有副本为单个数据结构，将依赖性与任务分离（“排除在并发计算之外”）。通常累计是一个*归约*运算，通过重复地应用一个二元运算，如加或乘，将数据元素的一个集合最终归约为单个元素。

  依赖性的处理方式可以更详细地描述如下：把累计中所使用的数据结构的副本复制到每一个 UE 上。初始化每一个副本（在归约情形中，把它初始化为二元运算的单位元，例如，对于加法来说，单位元为 0，对于乘法来说，单位元为 1）。然后每一个任务对它的局部数据结构进行累计运算，从而消除了共享数据的依赖性。当所有的任务完成时，每一个 UE 上的局部数据结构被组合成最终的全局结果（在归约情形中，通过重新应用二元运算来实现）。作为一个示例，考虑下面对数组 f 的元素累加的循环：

```
for(int i = 0; i< N; i++){
   sum = sum + f(i);
}
```

这是循环迭代间的依赖性，但是如果我们识别出循环体只不过是对某个简单标量元素的累加，则它可以被当作一个归约来处理。

归约非常普遍，以至于 MPI 和 OpenMP 都以 API 的形式提供了对它的支持。6.4.2 节更详细地讨论归约方面的内容。

- **其他依赖性**。如果共享数据无法被排除在任何任务之外，并且又需要被任务读和写，
67 则必须在任务内部显式地处理数据依赖性。通过什么方式完成该任务，使得结果正确并且获得一个可以接受的性能将是*共享数据*模式中的主题。

**调度**。其他需要考虑的关键因素是调度，即任务被分配给 UE 并被调度执行的方式。在调度中，负载平衡（如第 2 章所述）是一个调度中需要重点考虑的因素。当一个设计能够平衡 PE 间的计算负载时，它的执行效率要高于 PE 间计算负载不平衡的情况。图 4-3 演示了这个问题。

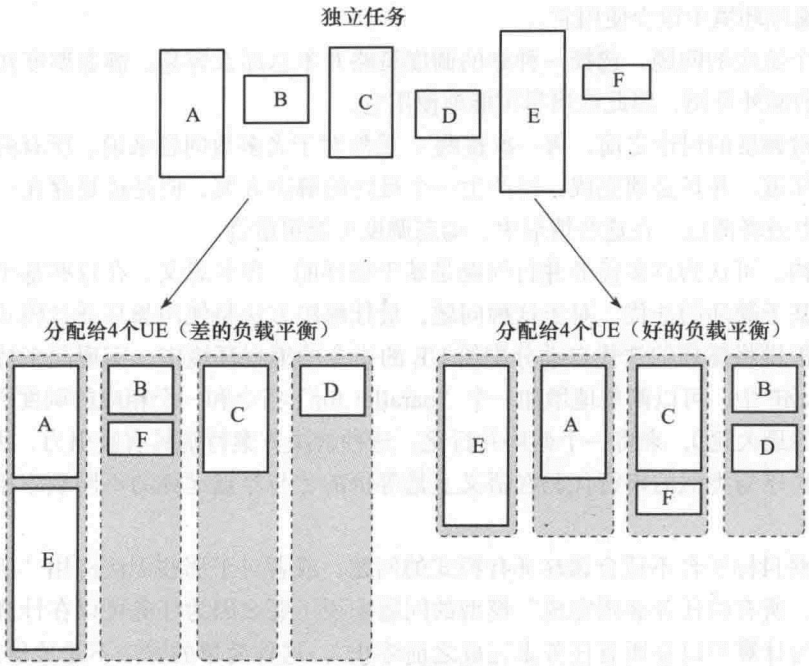



图 4-3 好的负载平衡和差的负载平衡的比较

并行算法中使用了两类调度：静态调度，其中任务在 UE 间的分布在计算开始时就已经确定，并且不再改变；动态调度，其中任务在 UE 间的分布随着计算的进行而发生改变。

在静态调度中，把任务组合成任务块，然后分配给 UE。任务块的大小是可调的，所以，每一个 UE 将花费几乎相等的时间来完成分配给它的任务。大多数应用中使用了静态调度，可从 UE 中获得的计算资源在计算的整个过程中都是可以预知并且稳定的，大多数情形中，UE 都是一致的（即计算系统是同构的）。如果完成每一个任务所需要的时间狭窄地分布在一个平均数附近，则每个任务块的大小应当与 UE 的性能成比例（因此，在一个同构系统中，所有任务块大小相同）。当完成任务所需的工作量差别很大时，静态调度仍然非常有用，但是现在分配给 UE 的任务块的数目必须要远大于 UE 的数目。通过以一种轮循（round-robin）方式来处理任务（像在一组玩家间发一副纸牌一样），将静态地平衡负载。 68

动态调度应用于两种情况：①与每一个任务相关的工作量差别很大并且是不可预知的。② UE 的性能差别很大，并且是不可预知的。动态负载平衡所使用的最常见方法是定义一个任务队列，该队列供所有的 UE 使用；当一个 UE 完成它的当前任务并且准备处理更多的工作时，它从任务队列中移除一个任务。性能较强的 UE 和那些获得轻量级任务的 UE 将会更频繁地访问队列，从而被分配更多的任务。

另一种动态调度策略是使用*工作窃取法*（work stealing），工作方式如下所述。在计算开始时，把任务分布到所有 UE 上。每一个 UE 具有自己的工作队列。当工作队列为空时，UE 将设法从其他 UE 的任务队列中窃取工作（这里，其他的 UE 通常是随机选择的）。多数情况下，这可以获得一个最佳的动态调度，而不会带来因维护一个全局队列而产生的开销。在提供这种构造支持的编程环境或者工具包中，如 Cilk[BJK+96]、Hood [BP99] 或者 FJTask 框架 [Lea00b，Lea]，使用这种方法非常简单。但是，许多常用的编程环境不提供这种支持，如 OpenMPI、MPI 或者 Java（不支持类似 FJTask 框架），因为使用这种方法将显著增加复杂性，

所以，这些编程环境中很少使用它。

针对一个给定的问题，选择一种好的调度策略并不总那么容易。静态调度在并行计算期间产生最少的额外开销，因此应当尽可能地使用它。

在结束对调度的讨论之前，再一次提醒：虽然对于大多数问题来说，所有的任务在计算开始时已经知道，并且必须完成，以产生一个最终的解决方案，但是还是存在一些问题不满足其中的一点或者两点。在这些情形中，动态调度可能更适合。

**程序结构**。可认为许多任务并行问题是基于循环的。顾名思义，在这些基于循环的问题中的任务都基于循环的迭代。对于这种问题，最佳解决方法是使用*循环并行*模式。这种模式在那些能够提供将循环的迭代自动分配给 UE 的指令的编程环境中，实现起来特别容易。例如，在 OpenMP 中，可以简单地增加一个“parallel for”指令和一个相应的调度子句（它能够使程序的效率最大化），来将一个循环并行化。这种解决方案特别具有吸引力，因此 OpenMP 保证了最终程序与类似的串行代码在语义上是等价的（与浮点运算的不同顺序相关的舍入错误范围之内）。

对于那些目标平台不适合*循环并行*模式的问题，或者对于那些无法应用“所有的任务初始时都已知，所有的任务必须完成”模型的问题来说（要么因为任务可以在计算期间动态创建，要么因为计算可以在所有任务未完成之前终止），这种简单的方法不是最佳选择。相反，充分使用*任务队列*才是最佳选择。当创建任务后，把它们放置到任务队列中，计算完成后，任务被 UE 从队列中移除。整个程序结构可以基于主 / 从（Master/Worker）模式，也可以基于 SPMD 模式。前者特别适用于那些需要动态调度的问题。

69

在所有的任务完成之前计算可以终止的情况下，需要特别小心，确保计算在应当结束时结束。如果定义一个*终止条件*，当条件为真时表明计算完成，或者所有任务都完成，或者某些其他条件满足（例如，一个任务已经找到了一个可接受的解决方案），则我们需要确保：①终止条件最终是可以满足的（例如，如果任务可以动态创建，则终止条件可能是所创建任务数目的最大值）；②当终止条件满足时，程序结束。如何确保后者将在 5.5 节和 5.4 节中讨论。

**常用术语**。可以应用这种模式的大多数问题分成如下两类。

*易并行*问题是那些任务间没有依赖性的问题。从动态影像的帧渲染到计算物理学中的统计采样，多数问题属于这一类。因为没有依赖性需要处理，所以主要工作集中于调度任务，以获得最大效率。在许多情形中，定义能够自动、动态地平衡 UE 间的负载的调度是可行的。

*复制数据*或归约问题是那些任务间的依赖性可以通过“将依赖性与任务相隔离”来处理的问题，如前面所述：在计算开始时复制数据，当终止条件满足时（通常，“所有的任务都完成”）组合所有的结果。对于这些问题，整个解决方案由三个阶段组成，开始时将数据复制到局部变量中，然后解决当前相互独立的任务（使用易并行问题中所使用的相同技术），最后重新组合所有的结果，得到一个最终的结果。

5. 示例

我们将考虑此模式的两个示例。第一个示例是图像构建示例，这是一个易并行问题。第二个示例构建于分子动力学示例之上，其中，分子动力学问题在第 3 章的好几节中都用过。

**图像构建**。在许多图像构建问题中，图像中的每一个像素独立于其他所有的像素。例如，

考虑知名的 Mandelbrot 集合 [Dou86]。这个著名图像的构建方式是根据二次递推关系对每一个像素进行着色：

$$Z_{n+1} = Z_n^2 + C \qquad (4\text{-}1)$$ 70

式中，$C$ 和 $Z$ 是复数，递推开始于 $Z_0$=$C$。在垂直轴上绘制 $C$ 的虚部，在水平轴上绘制 $C$ 的实部。如果递推关系收敛于一个稳定的值，则每一个像素的颜色是黑色，或者根据递推关系的发散速度对所有的像素进行着色。

在最底层，任务是对单个的像素进行更新。首先考虑在一个由以太网连接的 PC 集群上计算这个任务集合。该集群是一个粗粒度系统，即通信率相对于计算率是非常慢的。为了补偿慢速网络所产生的开销，任务的规模要足够大。对于这个问题，可能意味着计算图像的一整行。根据每一行中离散像素数目的不同，计算每一行所含的工作量是有差别的。但这个差别是适度的，紧密分布在一个平均值周围。因此，在任务数目远比 UE 数目多的情况下，静态调度将很可能在节点间给出一个高效的统计负载平衡。应用这种模式的剩余步骤是为程序选择一个整体结构。在一台使用 OpenMP 的共享内存机器上，第 5 章描述的循环并行模式是一个很好的选择。在一个运行 MPI 的工作站网络上，SPMD 模式（将第 5 章中描述）是一个合适的选择。

在转向下一个示例之前，我们考虑另外一个目标系统——集群，其中它的节点是异构的，也就是说，某些节点远比其他节点快。另外，假设在工作调度时，每一个节点的速度是未知的。因为计算图像的一行所需要的时间既依赖于该行本身，又依赖于计算它的节点，所以动态调度是最佳选择。这又需要使用一个通用的动态负载平衡策略，于是整个程序结构应当是基于主 / 从模式的。

**分子动力学。**对于第二个示例，我们考虑分子动力学问题中非化学键作用力的计算。这个问题描述在 3.1.3 节和 [Mat95，PH95] 中，第 3 章的所有模式中都使用了它。这个计算的伪码如图 4-4 所示。在这个例子中，物理上是不相关的，它隐藏在其他地方的代码中（这里仅是邻居列表和作用力函数的计算）。基本的计算结构是一个关于所有原子的遍历，然后对于每一个原子，又有另一个关于与其他原子交互的循环。当邻居列表确定时，每个原子交互的数目分别计算。该例程（这里没有显示）计算与预设的截断距离相等的半径内区域中的原子数目。邻居列表也被修改以说明牛顿第三定律：因为原子 `i` 对原子 `j` 的作用力是原子 `j` 对于原子 `i` 的反作用力，所以仅需要计算一半的原子潜在的交互即可。对于理解这个示例来说，理解示例的细节并不重要。重要的是，这使得对于不同的原子，每一个关于 `j` 的循环差别很大，因此，负载平衡问题变得非常复杂。实际上，针对该示例，必须理解的是计算作用力是一种开销很大的操作，并且每个原子交互的数目差别很大。因此，提前预测每一个关于 `i` 的迭代的计算量是很困难的。 71

计算作用力的每一部分是独立的，即每一个 (`i`,`j`) 对基本上是一个独立的任务。原子的数目趋向于以千为数量级，取该数目的平方后，所获得的任务数目远超过最大的并行系统所承载的任务数。因此，将一个任务定义为关于 `i` 的循环的迭代是非常便利的。但是，这些任务不是独立的：`force` 数组可供每一个任务读和写。从代码中观察发现，这些数组仅用于累加计算的结果。这样，整个数组可以复制到每一个 UE 中，任务完成后，局部副本再归约到一起。

```
function non_bonded_forces (N, Atoms, neighbors, Forces)

    Int const N // number of atoms

    Array of Real :: atoms (3,N) //3D coordinates
    Array of Real :: forces (3,N) //force in each dimension
    Array of List :: neighbors(N) //atoms in cutoff volume
    Real :: forceX, forceY, forceZ

    loop [i] over atoms

        loop [j] over neighbors(i)
            forceX = non_bond_force(atoms(1,i), atoms(1,j))
            forceY = non_bond_force(atoms(2,i), atoms(2,j))
            forceZ = non_bond_force(atoms(3,i), atoms(3,j))
            force(1,i) += forceX; force(1,j) -= forceX;
            force(2,i) += forceY; force(2,j) -= forceY;
            force(3,i) += forceZ; force(3,j) -= forceZ;
        end loop [j]

    end loop [i]
end function non_bonded_forces
```

图 4-4 一段典型的分子动力学代码中非化学键作用力计算的伪码

复制完成后，问题成为易并行问题，可以采用前面讨论的相同方法。我们会在 5.5 节、5.6 节和 5.4 节中重新回顾这个示例。对这些模式的选择通常还需要基于目标平台的特征进行考虑。

**知名应用。**在很多应用领域中，这种模式都非常有用。

许多光线追踪程序使用某种形式的分解，其中单独的任务对应于最终图像中的扫描线 [BKS91]。

使用协作语言（coordination language，例如 Linda）编写的应用程序是这种模式非常丰富的示例资源 [BCM+91]。Linda [CG91] 是一种简单的语言，仅由 6 种操作组成，这些操作读和
72 写一个相关（即内容寻址）共享内存，统称为元组空间。元组空间为各种共享队列和主 / 从算法的实现提供了一种很简便的方式。

并行计算化学应用也大量使用这种模式。在量子化学程序 GAMESS 中，关于两个电子体的循环使用了 TCGMSG 内的 `Nextval` 构造所隐含的任务队列，从而得到了并行化。距离几何程序 DGEOM 的一个早期版本使用了这种模式的主 / 从形式，从而实现了并行化。这些示例在 [Mat95] 中描述过。

并行遥测处理机（Parallel Telemetry Processor，PTEP）[NBB01] 由 NASA 开发，作为行星探测飞行器或月球登陆车数据的下行处理系统，也使用了这种模式。系统由 Java 实现，但可以与由其他语言开发的组件进行合并。对于每一个进入的数据包，系统确定是哪一个仪器产生了该数据，然后将该数据放入一个由多个处理步骤组成的串行流水线中处理。因为进入的数据包是相互独立的，所以每一个数据包的处理可以并行进行。

## 4.5 分治模式

1. 问题

假设问题使用串行的分治策略进行了描述，如何挖掘潜在的并发性？

2. 背景

分治策略在很多串行算法中得到了应用。利用这种策略，问题被划分为许多较小的子问题，独立地求解每一个子问题，并将所有的子问题解决方案合并为整个问题的解决方案，从而求解整个问题。子问题可以直接求解，或者可以再使用相同的分治策略求解，这样产生一个整体的递归程序结构。

对于大量的计算密集型问题来说，这个策略是非常有价值的。对于很多问题来说，它们的数学描述可以很好地映射到一个分治算法。例如，著名的快速傅里叶变换算法 [PTV93] 实质上是关于离散傅里叶变换的双嵌套循环到一个分治算法的映射。鲜为人知的是，计算线性代数中的许多算法，例如 Cholesky 分解 [ABE+97，PLA]，也能够被很好地映射到分治算法。

这种策略中的潜在并发性并不难发现，因为子问题可以独立地求解，所以它们可以同时进行计算。图 4-5 演示了这种策略和潜在的并发性。注意，每一次“划分”使可用的并发性加倍。尽管一个分治算法中的并发性是显然的，但有效挖掘它所需要的技术并不总是显而易见的。 73

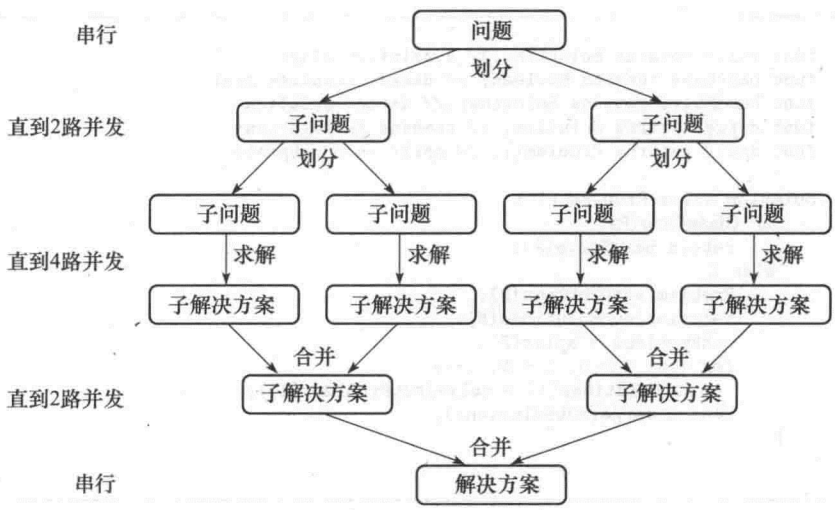



图 4-5 分治策略

3. 面临的问题

- 传统的分治策略是算法设计中广泛采用的一种方法。基于显而易见的并发性，串行分治算法可以很容易地并行化。
- 如图 4-5 所示，在程序的不同阶段，可供挖掘的并发性数量是不同的。在递归的最外层（初始的划分和最终的合并），仅有少量或没有可挖掘的并发性，并且子问题也包含划分和合并部分。Amdahl 定律（见第 2 章）告诉我们，一个程序的串行部分可以极大地制约通过添加更多的处理器所获得的加速比。这样，如果相比于基本的计算量，划分和合并计算并不能忽略，则使用这种模式的程序可能无法利用大量处理器的优点。此外，如果具有多层递归，任务数量可能增长得非常快，任务管理的开销可能远远超过并发执行所带来的好处。
- 在分布式存储系统中，子问题可以在一个 PE 上生成，而在另一个 PE 上执行，这需要将数据和结果在 PE 间移动。如果与计算相关的数据量（即参数集的大小和每一个子

问题结果的大小）较小，则算法会比较高效。否则，大的通信开销将降低性能。

- 在分治算法中，随着计算的进行动态地创建任务，在某些问题中，由此产生的“任务图”将有一个不规则和与数据相关的结构。如果发生这种情形，则应当利用动态负载平衡的解决方案。

74

4. 解决方案

一个串行分治算法的结构如图4-6所示。这个结构的基础是一个递归调用函数（slave()），该函数驱动解决方案中的每一步。在solve内部，问题或者划分为较小的子问题（使用split()函数），或者直接求解（使用baseSolve()函数）。典型的策略是将递归持续下去，直到子问题足够简单，可以直接求解为止，通常子函数中只有少数几行代码。但是，如果采用如下观点，效率将得到提高，即baseolve()函数应当在以下两种情况下调用：①当执行进一步的划分和合并的开销显著降低性能时；②当问题的规模对于目标系统来说是最优时（例如，当baseSolve()函数所需要的数据正好适合缓存的大小时）。

```
func solve returns Solution; // a solution stage
func baseCase returns Boolean; // direct solution test
func baseSolve returns Solution; // direct solution
func merge returns Solution; // combine subsolutions
func split returns Problem[]; // split into subprobs

Solution solve(Problem P) {
    if (baseCase(P))
        return baseSolve(P);
    else {
        Problem subProblems[N];
        Solution subSolutions[N];
        subProblems = split(P);
        for (int i = 0; i < N; i++)
            subSolutions[i] = solve(subProblems[i]);
        return merge(subSolutions);
    }
}
```

图4-6 分治算法的串行伪代码

通常情况下，当子问题可以独立求解时（因此可以并发地求解），分治问题中的并发性很容易被发现。通过将任务定义为solve()函数的一个调用，该串行分治算法可以直接地映射到一个任务并行算法，如图4-7所示。注意这个设计的递归属性，其中每一个任务实际上都动态地产生，然后由每一个子问题吸收一个任务。

在递归的某一层，子问题需要的计算量可能变得很小，以至于不值得创建一个新任务来求解它。在这种情况下，当子问题小于某个阈值时，在递归的较高层创建新任务的混合程序将转向串行求解方案，这样程序将更高效。像后面所讨论的一样，根据具体的问题和可用的PE数目，在选择这个阈值时，需要做出某种权衡。这样，在程序设计中，“粒度块”大小易于改变是一个非常好的主意。

75

**将任务映射到UE或者PE。**从概念上讲，这种模式遵循一种简单的派生/聚合（fork/join）方法（参考5.7节）。一个任务分割问题，然后派生新任务来计算这些子问题，然后等待，直到所有的子问题计算完成，最后与子任务结果合并，进而完成任务。

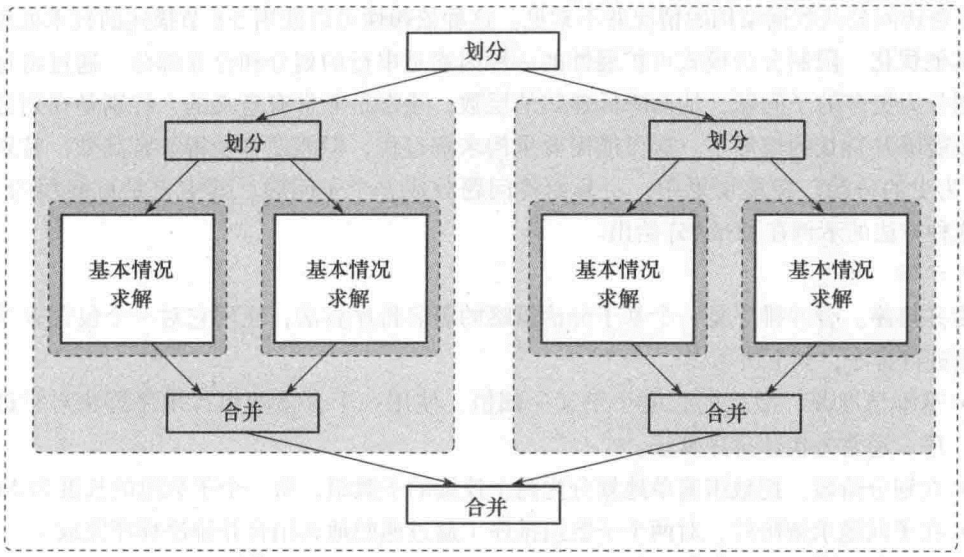



图 4-7 并行化分治策略（每个虚线框表示一个任务）

最简单的情形是当划分阶段产生计算量大小相同的子问题时。此时派生 / 聚合策略的一个简单实现是，将每一个任务映射到一个 UE 上，当活动的子任务数与 PE 数相同时，停止递归。

在许多情形中，问题是不规则的，因此最好创建多个细粒度的任务，使用主 / 从结构将任务映射到执行单元上。这个方法的详细实现在 5.5 节中讨论。基本思想是在概念上维持一个任务队列和一个 UE 池。当划分某个子问题时，把新的任务放置到队列中。当某个 UE 完成一个任务时，它从队列中获得另外一个任务。采用这种方式，所有的 UE 仍趋向于保持繁忙，该解决方案展现了很好的负载平衡。细粒度的任务以较多的任务管理开销为代价，获得较好的负载平衡。

许多并行编程环境直接支持派生 / 聚合结构。例如，在 OpenMP 中，我们可以容易地产生一个并行应用程序，方法是将图 4-6 中的 for 循环转换为 OpenMP 的一个 parallel for 结构。然后将并发地求解子问题，OpenMP 运行时环境负责处理线程的管理。遗憾的是，这种技术仅在 OpenMP 实现中有效，OpenMP 支持并行区域的嵌套。当前，仅有少数 OpenMP 实现了这些功能。将 OpenMP 扩展到更好的地址递归并行算法是 OpenMP 社区中一个很热的研究领域 [Mat03]。在将来的 OpenMP 规范中很可能采用的一个建议是，添加一个显式的任务队列结构，以支持递归算法的表达式 [SHPT00]。 76

Java 的 FJTask 框架 [Lea00b，Lea] 为派生 / 聚合程序提供了支持，包括一个支持实现的线程池。该程序包中提供了分治策略的几个示例程序。

**通信开销**。因为其他任务是由一个最顶层的任务动态地产生的，所以一个任务可以在产生它的 PE 之外的其他 PE 上执行。在一个分布式内存系统中，层次较高的任务通常具有求解它的整个问题所需要的数据，相关数据必须移动到子问题的 PE 中，而结果则需要移动到它的源处。这样，就需要考虑如何高效地表示参数和结果，同时考虑在计算开始时复制某些数据是否有意义。

**处理依赖性**。在使用分治策略构成的大多数算法中，每一个子问题可以独立地求解。子

问题需要访问公共数据结构的情况并不常见。这种依赖性可以使用 5.8 节描述的技术处理。

**其他优化。**限制分治模式可扩展性的一种因素是串行的划分和合并部分。通过将每一个问题划分为更多的子问题，从而降低递归的层数，通常是非常有意义的，特别是在划分和合并阶段能够并行化的情形下。这可能需要重构求解过程，但能使它变得非常高效，特别是在“深度为 1 的分治”极端情形中，一开始将问题分成 $P$ 个子问题，其中 $P$ 是可利用的 PE 数目。这种方法的示例在 [Tho95] 给出。

5. 示例

**合并排序。**合并排序是一个基于分治策略的著名排序算法，应用它对一个包含 $N$ 个元素的数组进行排序，如下所示。

- 基本情况是，数组的长度小于某一阈值。使用一个合适的串行排序算法对它进行排序，通常是快速排序算法。
- 在划分阶段，把数组简单地划分为两个连续的子数组，每一个子数组的长度为 $N/2$。
- 在子问题求解阶段，对两个子数组排序（通过递归地调用合并排序程序完成）。
- 在合并阶段，把两个已排序的子数组重新合并为一个已排序的数组。

77 通过并行地执行两个递归的合并排序，这个算法可以很容易地并行化。

5.7 节将更详细地介绍这个示例。

**矩阵对角化。**Dongarra 和 Sorensen（[DS87]）描述了对一个对称三角矩阵 $\boldsymbol{T}$ 对角化的并行算法（计算特征向量和特征值）。问题是寻找一个矩阵 $\boldsymbol{Q}$，使得 $\boldsymbol{Q}^{\mathrm{T}}\cdot\boldsymbol{T}\cdot\boldsymbol{Q}$ 是一个对角矩阵，所使用的分治策略如下（忽略了数学细节）：

- 基本情况是一个小的矩阵，它被串行地对角化。
- 划分阶段由寻找矩阵 $\boldsymbol{T}$ 和向量 $\boldsymbol{u}$、$\boldsymbol{v}$ 组成，使得 $\boldsymbol{T}=\boldsymbol{T}+\boldsymbol{u}\boldsymbol{v}^{\mathrm{T}}$，并且 $\boldsymbol{T}$ 的形式如下：

$$\begin{bmatrix}\boldsymbol{T}_1 & 0\\ 0 & \boldsymbol{T}_2\end{bmatrix}.$$

  其中，$\boldsymbol{T}_1$ 和 $\boldsymbol{T}_2$ 是对称的三角矩阵（可以递归地调用相同的步骤将它们对角化）。
- 合并阶段将对角化矩阵 $\boldsymbol{T}_1$ 和 $\boldsymbol{T}_2$ 重新组合为对角化矩阵 $\boldsymbol{T}$。

详情可参考 [DS87] 或 [GL96]。

**知名应用。**介绍算法的书籍中有许多基于分治策略的算法示例，其中大多数都可以利用这种模式进行并行化。

一些算法频繁地利用这种策略进行并行化，包括：用在 $N$-body 仿真中的 BarnesHut [BH86] 算法和 Fast Multipole [GC90] 算法；一些信号处理算法，例如，离散傅里叶变换；带状和三对角线性系统中的一些算法，例如，ScaLAPACK 程序包 [CD97，Sca] 中的一些算法；计算几何中的一些算法，例如，凸包和最近邻域。

在 FLAME 项目 [CGHvdG01] 中大量使用了分治模式。这是一个在递归算法中重塑线性代数问题的大项目，有双重动机。第一，在数学上，这些算法是自然递归的，事实上，这些算法在数学上的讨论大多也都是递归的；第二，这些递归算法在产生可移植且针对现代微处理器缓存体系结构高度优化的代码方面，已经被证明是特别高效的。

6. 相关模式

一个基于串行分治策略的算法并不意味着它必须利用分治模式来进行并行化。分治模式

的特点是任务的递归安排，从而导致变化的并发性和潜在的高系统开销，因为管理递归的开销是非常昂贵的。但是，如果子问题的递归分解可以重用，然后使用某些其他的模式（例如，*几何分解模式*或*任务并行模式*）进行实际的计算，则进行递归分解可能更有效。例如，第一个产品级的分子动力学程序中运用了快速多极子（fast multipole）算法 PMD [Win95]，尽管初始的快速多极子算法使用的是分治策略，但使用*几何分解模式*对快速多极子算法进行并行化。之所以这样做，是因为针对每个原子的配置进行了多次多极子计算。 78

## 4.6 几何分解模式

1. 问题

算法如何围绕着一个已经分解为多个同时可更新的"块"的数据结构进行组织？

2. 背景

最好把许多重要问题理解为核心数据结构上的操作序列。在计算中可能存在其他的工作，但是通过理解核心数据结构是如何更新的，可以有效理解整个计算。对于这种问题，表示并发性的最好方式是根据核心数据结构的分解（并发性的这种形式有时称为域分解，或粗粒度数据并行性）。

这些数据结构的构建方式是算法的基础。如果数据结构是递归的，并行性分析必须考虑这种递归。对于递归数据结构，*递归数据*模式和*分治*模式很可能是两种候选模式。对于数组和其他线性数据结构，我们通常可以通过将数据结构分解为多个连续的子结构，将问题归约为多个潜在的并发部件，这类似于将一个几何区域划分为多个子区域的方式，即所谓的*几何分解*。对于数组，这种分解可以沿着一维或多维进行，分解的子数组通常称为数组块（block）。对于子结构或子区域，我们将使用术语块（chunk）表示它们，该术语还可以表示一些更普通的数据结构，如图（graph）。

数据到块的分解暗含着更新操作到任务的分解，其中每一个任务表示一个块的更新，并且这些任务可以并发地执行。如果计算是严格局部的，即所需要的所有信息都在块内，则该并发性是易并行的，应当使用较简单的*任务并行*模式。但是，在许多情形中，更新操作需要其他块中的信息（我们称这些块为*邻居块*，即这些块中包含的数据是初始全局数据结构附近的数据）。在这些情形中，信息必须在块间通过共享来完成更新。 79

**示例：网格计算程序。**问题是模拟 1D 热扩散（即沿着一个无限狭窄管道的热扩散）。初始时，整个管道处于一个稳定和固定的温度。在 0 时刻，我们将管道两端设置为不同的温度，在整个计算期间，这两个温度保持恒定。然后随着时间的推移，我们计算管道其他部分的温度是如何变化的（我们所期望的是，温度从管道的一端到另一端保持平滑的梯度）。在数学上，这个问题是求解一个表示热扩散的 1D 微分公式：

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} \tag{4-2}$$

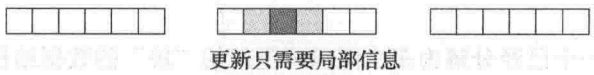
所使用的方法是离散化问题空间（$U$ 表示一个一维数组，并计算一个离散时间步序列的值）。当计算每一个时间步的值时，将输出它的值，因此仅需要为 $U$ 保存两个时间步的值。将该值存储在两个数组 `uk`（$U$ 在时间步 $k$ 的值）和 `ukp1`（$U$ 在时间步 $k$+1 的值）。在每一个时间步，需要为数组 `ukp1` 中的每一个点计算结果：

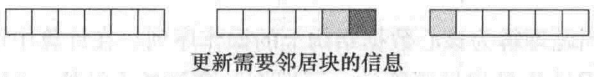
```
ukpl[i]=uk[i]+(dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
```

变量 dt 和 dx 分别表示离散时间步间的间隔和离散点间的间隔。

我们发现，正在计算的是在每一点处变量 ukpl 的一个新值，计算需要该点左边邻居点和右边邻居点的数据。

可以通过将数组 uk 和 ukpl 划分为连续的子数组（前面描述的块），为这个问题设计一个并行算法。这些块可以并发地操作，从而给出可挖掘的并发性。注意，我们处于这样一种情形之中，其中某些元素可以使用块内部的数据进行更新，而某些元素却需要通过访问邻居块中的数据进行更新，如图 4-8 所示。




80 图 4-8 热扩散问题中的数据依赖性。深色块表示要更新的块，浅色块表示更新需要的数据

**示例：矩阵乘法程序。**考虑两个方阵的乘法（即计算 $C=A\times B$）。像 [FJL+88] 中讨论的一样，矩阵可以划分为多个块。矩阵乘法定义中的求和同样组织为块，这使得我们可以编写一个关于块的矩阵乘法公式：

$$C^{ij}=\sum_k A^{ik}\cdot B^{kj} \tag{4-3}$$

在求和中的每一步，计算矩阵积 $A^{ik}\cdot B^{kj}$，并将它累加到矩阵和中。

这个公式立即给出了一个利用几何分解模式的解决方案，即该解决方案中的算法基于将数据结构划分为多个能够并发操作的块（这里是方形块）。

为了更加清晰地了解这个问题，考虑 3 个矩阵都划分为矩阵块的情形，其中每一个任务“拥有” $A$、$B$、$C$ 相应的矩阵块。每个任务将运行关于 $k$ 的求和操作来计算它负责的 $C$ 矩阵块，如果需要，任务还可以从其他任务接收矩阵块。在图 4-9 中，我们演示了两个不同的步骤，在这个过程中，显示了一个正在更新的矩阵块（深色块）和所需要的矩阵块（浅色块），其中，$A$ 矩阵的矩阵块被传递一行，而 $B$ 矩阵的矩阵块被传递一列。

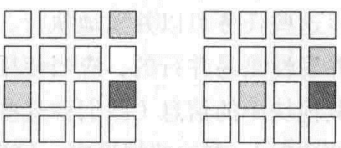



图 4-9 矩阵乘法问题中的数据依赖性。在两个步骤中，深色块表示正在更新的块（$C$）；浅色块表示更新 $C$ 所需要的 $A$ 的块（行）和 $B$ 的块（列）

3. 面临的问题

- 为了挖掘问题中的潜在并发性，我们必须将分解的数据结构的块分配给 UE。理想情况下，我们想采用一种简单、可移植、可扩展和高效的方式完成这个工作。然而，如 4.1 节所述，这些目标可能相互冲突。需要考虑的关键是如何平衡负载，即确保所有的 UE 具有近似相等的工作量。
- 我们还必须确保每一个块更新时所需要的数据在需要时能够存在。这个问题与任务并行模式中管理数据依赖性的问题类似。在设计时，我们必须牢记简单性、可移植性、可扩展性和效率有时是相互冲突的目标。

4. 解决方案

适合这种模式的问题的设计包含如下关键因素：全局数据结构被分割为多个子结构或“块”（数据分解），确保每一个任务能够访问它所需要的所有数据，以完成它负责的块的交互操作和块的更新操作；采用一种可获得良好性能的映射方式完成块到任务的映射（数据分配和任务调度）。

**数据分解。**数据分解的粒度对程序的性能具有重要的影响。在粗粒度分解中，存在较少数目的大块。这将产生数目较少的大消息，这样可以极大地降低通信开销。另一方面，细粒度分解将导致数目较大的较小块，在许多情形中，这将产生比 PE 数目还多的块，从而导致数目较大的较小消息（因此增加通信开销），但是它极大地方便了负载平衡。

尽管在某些情形下可以通过数学方法为数据分解推导出一个最优的粒度，但程序员通常对某个范围的块大小进行实验，根据经验确定给定系统的最佳粒度。当然，这依赖于 PE 的计算性能和通信网络的性能特征。因此，所实现的程序的粒度应当由参数来控制，这些参数在编译时或运行时很容易改变。

任务分解的块的形状也能够影响任务间所需要的通信量。通常，任务间共享的数据被限制为块的边界处的数据。在这种情形中，共享信息量随着块的表面积的大小而缩放。因为计算量也随着一个块内的点的数目而缩放，所以它随着该区域的容积而缩放。可以开发利用这种表面积对容积（surface to volume）作用来最大化计算与通信的比例。因此，通常需要进行较高维的分解，例如，考虑一个 $N \times N$ 矩阵分解为 4 个块的两种不同分解模式。在第一种情形中，我们将问题分解为 4 列的块，每一块的大小为 $N \times (N/4)$。在第二种情形中，我们将问题分解为 4 个方块，大小为 $(N/2) \times (N/2)$。对于块的列分解，表面积为 $2N+2(N/4)$，或者 $5N/2$。对于方块分解，表面积是 $4(N/2)$，或者 $2N$。因此，对于方块分解来说，需要交换的数据总量较少。

在某些情形中，最好的分解形状受其他一些因素影响。例如，在某个情形中，对于较低维的分解来说，可以很容易地使现存的串行代码被重用，并且潜在的性能增加不值得重新编写代码。此外，这种模式的一个实例可以作为一个串行步骤用在一个较大的计算中。如果在相邻步骤中使用的分解与单独使用这种模式所使用的最佳分解不同，则可能值得也可能不值得为这个步骤重新分配数据。特别是在分布式存储系统中，这是一个非常重要的问题。在分布式存储系统中，重新分配数据将需要大量的通信，整个计算将延迟。因此，数据分解的决策必须考虑串行代码重用的能力和计算中与其他步骤交互的需要。注意，这些考虑因素所产生的分解在其他情形中可能不是最理想的。

81 ~ 82

通过复制一个块中的数据更新需要的非局部数据，通常可以更加高效地管理通信。例如，如果数据结构是一个表示网格中的点的数组，并且更新操作使用网格中的一个局部邻近点，则一种常用的通信管理技术是在该块的数据结构之外环绕一个影像边界（ghost boundary），用于包含邻近块边界处的数据的副本。因此现在每一块具有两部分：UE 所拥有的主副本（它将被直接更新）和零个或多个影像副本（也称为阴影副本）。这些影像副本有两个优点。首先，它们的使用可以合并通信而使得通信的消息数量变少，而消息内容变多。在延迟敏感的网络中，这样可以极大地降低通信开销。第二，影像副本的通信可以与那些不依赖该影像副本内的数据的数组部分的更新相重叠（即可以并发地执行）。实质上，这将通信开销隐藏在有用的计算之下，从而降低了可见的通信开销。

例如，在之前讨论的网格计算的示例中，每一个块的每一端将被扩展一个单元。这些特别的单元将用作块的边界单元的影像副本。图 4-10 演示了这种策略。

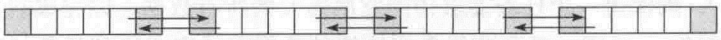


图 4-10 影像边界的一个数据分布（阴影单元是影像副本，箭头从主副本指向对应的副副本）

**交换操作**。正确使用这种模式的关键因素是，确保更新操作所需要的非局部数据在使用之前可以获得。

如果所需要的所有数据在更新操作开始之前都能够提供，最简单的方法是在更新开始之前完成整个交换，将所需要的非局部数据存储在一个专门设计的局部数据结构中（例如，网格计算中的影像边界）。无论是使用复制还是使用消息传递，这种方法使用起来都相对简单。

83 计算和通信能够重叠的更多先进的方法也是存在的。如果更新所需要的数据在初始时无法获得，这样的方法是必需的，并且在其他的情形中也能够提高性能。例如，在网格计算的例子中，影像单元的交换和内部区域中单元（它不依赖于影像单元）的更新可以并发进行。交换完成后，可以更新边界层（这些值不依赖于影像单元）。在通信和计算并行进行的系统中，这种方法节约的时间是显著的。这是并行算法一个非常通用的特征，标准通信 API（如 MPI）包含消息传递例程的整个类，从而完成计算和通信的重叠。这些将在附录 B 中更加详细地讨论。

交换操作实现方式的底层细节对效率具有重大影响。程序员应当挑选出通信模式的最佳实现，以用于程序中。例如，在许多应用程序中，消息传递库（如 MPI）中许多集合通信例程是非常有用的。这些例程已经被精心优化过了，并且所使用的技术超出了许多并行程序员的能力（6.4.2 节会讨论其中的一些技术），因此应当尽可能地使用它们。

**更新操作**。通过并发地执行相应的任务完成数据结构的更新（每一个任务负责数据结构一个块的更新）。如果所需要的所有数据在更新操作开始之前已提供，并且这些数据在更新过程中都不修改，则并行化是较简单的，并且也可能更有效。

如果所需要的信息交换在更新操作开始之前已经执行，则更新本身是很容易实现的，它实质上与一个与之相当的串行程序中的更新完全一致，特别是在如何表示非局部数据方面已经做了很好的选择的情况下。

如果交换操作和更新操作重叠，则需要特别小心以确保更新正确地执行。如果系统支持与通信系统很好集成的轻量级线程，则可以在单个任务内通过多线程来完成重叠，其中一个线程负责计算，另外一个线程处理通信。在这种情形中，线程之间的同步是必要的。

在某些系统（如 MPI）中，通过配合通信原语来支持非阻塞通信：一个通信原语用于启动通信（无阻塞），而另一个（阻塞）用于完成操作和使用结果。为了最大化重叠，通信应当尽可能早地启动，而且应当尽可能晚地结束。有时，为了获得更多的重叠，允许在不改变算法语义的前提下，对操作重新排序。

**数据分布和任务调度**。为适合这种模式的问题设计一个并行算法的最后一步是确定如何将任务集（每个任务对应于每块的更新）映射到 UE 上。于是可以认为每个 UE“拥有”关于块的一个集合和它们所包含的数据。因此，对于在 UE 间分配数据，我们具有一个双重策略：

84 首先将数据分解为多个块，然后把这些块分给 UE。该方案足够灵活地表示在 UE 间分布数据

的各种各样的策略。

在最简单的情形中，可以给每一个任务静态地分配一个单独的 UE，然后所有的任务可以并发地执行，并且实现交换操作所需要的任务间协作也非常简单。当任务的计算时间是均匀的，并且被实现的交换操作可以重叠每一个任务内的计算与通信时，这种方法是最适合的。

但是，在某些情况下，这种简单方法可能导致较差的负载平衡。例如，考虑一个线性代数问题。在该问题中，随着计算的进行，矩阵的元素被逐渐地消除。在计算的早期，矩阵的所有行和列有大量的元素待处理，基于将整个行或列分配给 UE 的分解是高效的。但是，在计算的后期，行和列变得很稀疏，每一行的任务量变得不均匀，UE 间的计算负载变得很不平衡。解决方案是将问题分解为更多的块，并采用周期或块周期分配方式将它们分配到 UE 上（周期或块周期分配将在 5.10 节中讨论）。然后，当块变得稀疏时（可能性非常大），存在其他的非稀疏块可以用任意给定的 UE 去处理，负载将变得非常平衡。为了很好地实现负载平衡，一个经验法则是任务的数目大约是 UE 数目的 10 倍。

也可以使用动态负载平衡算法周期性地在 UE 间重新分配块，以改善负载平衡。这些工作带来的额外开销必须权衡改善的负载平衡带来的改进和增加的实现成本。另外，使用该方法的程序比那些只使用一种静态方法的程序要复杂得多。通常，应当首先考虑（静态）循环分配策略。

**程序结构**。对于这种模式的应用程序来说，整体程序结构通常使用*循环并行*模式或 SPMD 模式，具体选择哪种模式，很大程度上依赖于目标平台。*循环并行*模式和 SPMD 模式将在第 5 章中描述。

5. *示例*

我们在这种模式中包含两个示例：网格计算和矩阵乘法。应用几何分解模式的挑战在于最终程序的底层细节。因此，尽管到目前为止在程序中使用的这些技术还未完全介绍，但在本书的后面将介绍。我们将在本节中提供完整的程序，而不是解决方案的高层次描述。

**网格计算**。该问题的描述参见本节前面的内容。图 4-11 展示了一个简单的串行版本程序（忽略了某些细节），该程序求解了 1D 热扩散问题。这个程序非常简单，需要进一步解释的 85
程序细节是：在每一步计算完 ukp1 的新值之后，概念上，我们要做的就是将它们复制到 uk 中，以备下一次迭代使用。在每一步结束时，通过简单地将 uk 和 ukp1 指针交换来避免非常耗时的实际复制。这使得 uk 指向刚刚计算过的新值，ukp1 指向用于在下一次迭代中计算新值所需要的区域。

```
#include <stdio.h>
#include <stdlib.h>
#define NX 100
#define LEFTVAL 1.0
#define RIGHTVAL 10.0
#define NSTEPS 10000

void initialize(double uk[], double ukp1[]) {
  uk[0] = LEFTVAL; uk[NX-1] = RIGHTVAL;
  for (int i = 1; i < NX-1; ++i)
    uk[i] = 0.0;
```

图 4-11 串行热扩散程序

```
  for (int i = 0; i < NX; ++i)
    ukp1[i] = uk[i];
}

void printValues(double uk[], int step) { /* NOT SHOWN */ }

int main(void) {
  /* pointers to arrays for two iterations of algorithm */
  double *uk = malloc(sizeof(double) * NX);
  double *ukp1 = malloc(sizeof(double) * NX);
  double *temp;

  double dx = 1.0/NX;
  double dt = 0.5*dx*dx;

  initialize(uk, ukp1);

  for (int k = 0; k < NSTEPS; ++k) {

    /* compute new values */
    for (int i = 1; i < NX-1; ++i) {
      ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }

    /* "copy" ukp1 to uk by swapping pointers */
    temp = ukp1; ukp1 = uk; uk = temp;

    printValues(uk, k);
  }
  return 0;
}
```

86

图 4-11 （续）

这个程序组合了一个高层串行控制结构（时间步循环）与一个数组的更新操作，可以使用几何分解模式将其并行化。我们将展示使用 OpenMP 和 MPI 来完成这个程序的并行实现。

**OpenMP 解决方案。**使用了 OpenMP 和循环并行模式的程序的一个特别简单的版本，如图 4-12 所示。因为 OpenMP 是一种共享内存编程模型，所以没有必要显式地分解和分配两个主要数组（`uk` 与 `ukp1`）。线程的创建和线程间工作的分配由 `parallel for` 指令完成，如下所示：

```
#pragma parallel for schedule(static)
```

`schedule(static)` 子句将并行循环的迭代分解为多个连续的块，每一个块的大小近似相等，由每一个线程处理一个块。这种调度对于实现几何分解模式的循环并行程序来说非常重要。对于大多数几何分解问题来说（特别是网格程序），处理器缓存中的数据在被新的缓存行的数据替换之前，应当已使用很多次，这样才会获得好的程序性能。使用连续循环迭代中较大的块将增加所预取的缓存行中的多个值被利用的机会，并增加后续循环迭代在当前缓存中至少找到所需要数据的机会。

对于图 4-12 中的程序来说，所需要讨论的最后一个细节是安全复制指针所需要的同步。所有线程在为下一次迭代完成它们所操纵的指针交换之前，必须完成它们负责的工作，这一点非常重要。在这个程序中，根据并行循环结尾处隐含的栅栏自动地完成同步（参考 6.3.2 节）。

```
#include <stdio.h>
#include <stdlib.h>
#define NX 100
#define LEFTVAL 1.0
#define RIGHTVAL 10.0
#define NSTEPS 10000

void initialize(double uk[], double ukp1[]) {
  uk[0] = LEFTVAL; uk[NX-1] = RIGHTVAL;
  for (int i = 1; i < NX-1; ++i)
    uk[i] = 0.0;
  for (int i = 0; i < NX; ++i)
    ukp1[i] = uk[i];
}

void printValues(double uk[], int step) { /* NOT SHOWN */ }

int main(void) {
  /* pointers to arrays for two iterations of algorithm */
  double *uk = malloc(sizeof(double) * NX);
  double *ukp1 = malloc(sizeof(double) * NX);
  double *temp;

  double dx = 1.0/NX;
  double dt = 0.5*dx*dx;

  initialize(uk, ukp1);

  for (int k = 0; k < NSTEPS; ++k) {

    #pragma omp parallel for schedule(static)
    /* compute new values */
      for (int i = 1; i < NX-1; ++i) {
        ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
      }

    /* "copy" ukp1 to uk by swapping pointers */
    temp = ukp1; ukp1 = uk; uk = temp;

    printValues(uk, k);
  }
  return 0;
}
```

图 4-12 使用 OpenMP 的并行热扩散程序

图 4-12 中的程序在线程数目较少时能够很好地工作。但是，当涉及的线程数目较多时，循环 k 内部的线程的创建和销毁所带来的开销将是令人望而却步的。可以降低线程管理的开销，具体方式是将 parallel for 指令分解为单独的 parallel 指令和 for 指令，并将线程的创建移动到循环 k 的外部。图 4-13 展示了这种方法。现在因为整个 k 的循环位于一个并行区域的内部，所以我们必须对数据在线程间的共享更加谨慎。private 子句声明循环索引 k 和 i 为每一个线程的局部变量。但是，指针 uk 和 ukp1 是共享的，因此交换操作必须受到保护。最简单的方式是确保只有一个线程进行了交换操作。在 OpenMP 中，完成该工作的简单方式是将更新操作放置到一个 single 构造中。附录 A 详细地描述了这一点，遇到该结构的第一个线程将完成交换操作，而其他线程将在该结构的结尾处等待。

**MPI 解决方案**。关于该示例的一个基于 MPI 的程序如图 4-14 和图 4-15 所示。这个程序所使用的方法使用了一个对影像单元的数据分配和 SPMD 模式。

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define NX 100
#define LEFTVAL 1.0
#define RIGHTVAL 10.0
#define NSTEPS 10000

void initialize(double uk[], double ukp1[]){/* NOT SHOWN */}
void printValues(double uk[], int step) { /* NOT SHOWN */ }

int main(void) {
  /* pointers to arrays for two iterations of algorithm */
  double *uk = malloc(sizeof(double) * NX);
  double *ukp1 = malloc(sizeof(double) * NX);
  double *temp;
  int i,k;

  double dx = 1.0/NX;
  double dt = 0.5*dx*dx;

  #pragma omp parallel private (k, i)
  {
    initialize(uk, ukp1);

    for (k = 0; k < NSTEPS; ++k) {
      #pragma omp for schedule(static)
      for (i = 1; i < NX-1; ++i) {
        ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
      }
      /* "copy" ukp1 to uk by swapping pointers */
      #pragma omp single
      { temp = ukp1; ukp1 = uk; uk = temp; }
    }
  }
  return 0;
}
```

图 4-13 使用 OpenMP 的并行热扩散程序（这个版本具有较少的线程管理开销）

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#define NX 100
#define LEFTVAL 1.0
#define RIGHTVAL 10.0
#define NSTEPS 10000

void initialize(double uk[], double ukp1[], int numPoints,
        int numProcs, int myID) {
  for (int i = 1; i <= numPoints; ++i)
    uk[i] = 0.0;
  /* left endpoint */
  if (myID == 0) uk[1] = LEFTVAL;
  /* right endpoint */
  if (myID == numProcs-1) uk[numPoints] = RIGHTVAL;
  /* copy values to ukp1 */
  for (int i = 1; i <= numPoints; ++i) ukp1[i] = uk[i];
}

void printValues(double uk[], int step, int numPoints, int myID)
```

图 4-14 使用 MPI 的并行热扩散程序（续见图 4-15）

```
  { /* NOT SHOWN */ }
int main(int argc, char *argv[]) {
  /* pointers to arrays for two iterations of algorithm */
  double *uk, *ukp1, *temp;

  double dx = 1.0/NX; double dt = 0.5*dx*dx;

  int numProcs, myID, leftNbr, rightNbr, numPoints;
  MPI_Status status;

  /* MPI initialization */
  MPI_Init(&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myID); //get own ID

  /* initialization of other variables */
  leftNbr = myID - 1; // ID of left "neighbor" process
  rightNbr = myID + 1; // ID of right "neighbor" process
  numPoints = (NX / numProcs);
  /* uk, ukp1 include a "ghost cell" at each end */
  uk = malloc(sizeof(double) * (numPoints+2));
  ukp1 = malloc(sizeof(double) * (numPoints+2));

  initialize(uk, ukp1, numPoints, numProcs, myID);
/* continued in next figure */
```

图 4-14 （续）

```
/* continued from Figure 4.14 */

  for (int k = 0; k < NSTEPS; ++k) {

    /* exchange boundary information */
    if (myID != 0)
        MPI_Send(&uk[1], 1, MPI_DOUBLE, leftNbr, 0,
                                    MPI_COMM_WORLD);
    if (myID != numProcs-1)
        MPI_Send(&uk[numPoints], 1, MPI_DOUBLE, rightNbr, 0,
                                    MPI_COMM_WORLD);
    if (myID != 0)
        MPI_Recv(&uk[0], 1, MPI_DOUBLE, leftNbr, 0,
                                    MPI_COMM_WORLD, &status);
    if (myID != numProcs-1)
        MPI_Recv(&uk[numPoints+1],1, MPI_DOUBLE, rightNbr, 0,
                                    MPI_COMM_WORLD, &status);

    /* compute new values for interior points */
    for (int i = 2; i < numPoints; ++i) {
      ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }
    /* compute new values for boundary points */
    if (myID != 0) {
        int i=1;
        ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }
    if (myID != numProcs-1) {
        int i=numPoints;
        ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }

    /* "copy" ukp1 to uk by swapping pointers */
```

图 4-15 使用 MPI 的并行热扩散程序（续图 4-14）

```
    temp = ukp1; ukp1 = uk; uk = temp;
    printValues(uk, k, numPoints, myID);
  }

  /* clean up and end */
  MPI_Finalize();
  return 0;
}
```

图 4-15 （续）

每个进程被赋予一个数据域大小为 NX/NP 的块，其中 NX 是全局数据数组的总大小，NP 是进程的数目。为了简单起见，设 NX 可以被 NP 整除。

块的更新是简单的，实质上与串行代码中的方法一样。这个 MPI 程序的长度和较大的复杂性源于两个方面：第一，数据初始化更复杂，因为它必须考虑第一个块和最后一个块边缘处的数据值；第二，在关于 k 的循环内部需要消息传递例程来交换影像单元。

在附录 B 中可以找到消息传递函数的细节。简而言之，数据转发由两部分组成：一个进程负责一个发送操作，这需要指定包含发送数据的缓冲区；而另一个进程负责一个接收操作，这需要指定存放所接收到的数据的缓冲区。我们需要几个不同的发送和接收对，因为拥有数组最左块的进程不具有一个与之通信的左邻居，相似地，拥有数组最右块的进程不具有一个与之通信的右邻居。

如本节前面所述，可以使用非阻塞通信来重叠计算和通信，进一步修改图 4-14 和图 4-15 中的代码。与第一个网格计算 MPI 程序（即图 4-14 中的程序）相比，新程序的第一部分未发生变化。差别位于程序的第二部分，即包含主要计算循环的地方。该代码如图 4-16 所示。

```
/* continued */
  MPI_Request reqRecvL, reqRecvR, reqSendL, reqSendR; //needed for
                                                      // nonblocking I/O

  for (int k = 0; k < NSTEPS; ++k) {
    /* initiate communication to exchange boundary information */
    if (myID != 0) {
      MPI_Irecv(&uk[0], 1, MPI_DOUBLE, leftNbr, 0,
                MPI_COMM_WORLD, &reqRecvL);
      MPI_Isend(&uk[1], 1, MPI_DOUBLE, leftNbr, 0,
                  MPI_COMM_WORLD, &reqSendL);
    }
    if (myID != numProcs-1) {
      MPI_Irecv(&uk[numPoints+1],1, MPI_DOUBLE, rightNbr, 0,
                MPI_COMM_WORLD, &reqRecvR);
      MPI_Isend(&uk[numPoints], 1, MPI_DOUBLE, rightNbr, 0,
                MPI_COMM_WORLD, &reqSendR);
    }
    /* compute new values for interior points */
    for (int i = 2; i < numPoints; ++i) {
      ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }
    /* wait for communication to complete */
    if (myID != 0) {
      MPI_Wait(&reqRecvL, &status); MPI_Wait(&reqSendL, &status);
    }
    if (myID != numProcs-1) {
      MPI_Wait(&reqRecvR, &status); MPI_Wait(&reqSendR, &status);
```

图 4-16 使用了 MPI 的并行热扩散程序，其中重叠了通信与计算（续图 4-14）

```
    }
    /* compute new values for boundary points */
    if (myID != 0) {
      int i=1;
      ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }
    if (myID != numProcs-1) {
      int i=numPoints;
      ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
    }
    /* "copy" ukp1 to uk by swapping pointers */
    temp = ukp1; ukp1 = uk; uk = temp;

    printValues(uk, k, numPoints, myID);
  }
  /* clean up and end */
  MPI_Finalize();
  return 0;
}
```

图 4-16 （续）

虽然基本算法相同，但是通信方式的差别非常大。立即模式通信例程 MPI_Isend 和 MPI_IRecv 用来建立和启动通信事件。这两个函数（在附录 B 中有更详细的描述）立即返回。因为内部点不依赖于通信的结果，所以可以进行内部点的更新操作。直到通信完成才能调用函数，并使用通信事件的结果完成每个 UE 负责的块的边缘更新。在这种情形中，消息较小，因此这个版本的程序未必比第一个程序快。但很容易设想一种情形，其中引入了大的、复杂的通信事件，当消息在计算机网络间移动时，能够做一些有意义的工作，这样将带来显著的性能提升。

**矩阵乘法**。本节前面描述了矩阵乘法问题。基于将 N×N 矩阵分解为 NB*NB 个方块，图 4-17 给出了一个计算所需结果的简单串行程序。注释中的符号 block[i][j] 表示前面描述的第（i,j）个块。为了简化 C 语言代码，将矩阵表示为多个 1D 数组（内部按照行优先的排列顺序），并定义一个宏 blockstart，用于在其中某个 1D 数组中寻找一个子矩阵的左上角。我们省略了 initialize 函数（初始化矩阵 A 和 B）、printMatrix 函数（输出一个矩阵的值）、matclear 函数（清空矩阵——将所有的值设置为 0）和 matmul_add 函数（计算两个输入矩阵的矩阵乘积，并将结果加到输出矩阵上）的代码。这些函数的参数大多包括矩阵的维数，以及一个用于指示从矩阵一行的起点到下一行的起点的跨度，像作用在整个矩阵上一样，通过这些参数，我们能够把这些函数应用到子矩阵上。

```
#include <stdio.h>
#include <stdlib.h>
#define N 100
#define NB 4

#define blockstart(M,i,j,rows_per_blk,cols_per_blk,stride) \
  (M + ((i)*(rows_per_blk))*(stride) + (j)*(cols_per_blk))

int main(int argc, char *argv[]) {
  /* matrix dimensions */
  int dimN = N; int dimP = N; int dimM = N;
```

图 4-17 串行矩阵乘法

```
  /* block dimensions */
  int dimNb = dimN/NB; int dimPb = dimP/NB; int dimMb = dimM/NB;

  /* allocate memory for matrices */
  double *A = malloc(dimN*dimP*sizeof(double));
  double *B = malloc(dimP*dimM*sizeof(double));
  double *C = malloc(dimN*dimM*sizeof(double));

  /* Initialize matrices */

  initialize(A, B, dimN, dimP, dimM);

  /* Do the matrix multiplication */

  for (int ib=0; ib < NB; ++ib) {
    for (int jb=0; jb < NB; ++jb) {
      /* find block[ib][jb] of C */
      double * blockPtr = blockstart(C, ib, jb, dimNb, dimMb, dimM);
      /* clear block[ib][jb] of C (set all elements to zero) */
      matclear(blockPtr, dimNb, dimMb, dimM);
      for (int kb=0; kb < NB; ++kb) {
        /* compute product of block[ib][kb] of A and
           block[kb][jb] of B and add to block[ib][jb] of C */
        matmul_add(blockstart(A, ib, kb, dimNb, dimPb, dimP),
                   blockstart(B, kb, jb, dimPb, dimMb, dimM),
                   blockPtr, dimNb, dimPb, dimMb, dimP, dimM, dimM);
      }
    }
  }

  /* Code to print results not shown */

  return 0;
}
```

图 4-17 （续）

首先注意，可以重新排列循环，而不影响计算的结果，如图 4-18 所示。

```
/* Declarations, initializations, etc. not shown -- same as
   first version */

/* Do the multiply */

matclear(C, dimN, dimM, dimM); /* sets all elements to zero */

for (int kb=0; kb < NB; ++kb) {

  for (int ib=0; ib < NB; ++ib) {
    for (int jb=0; jb < NB; ++jb) {
      /* compute product of block[ib][kb] of A and
         block[kb][jb] of B and add to block[ib][jb] of C */
      matmul_add(blockstart(A, ib, kb, dimNb, dimPb, dimP),
                 blockstart(B, kb, jb, dimPb, dimMb, dimM),
                 blockstart(C, ib, jb, dimNb, dimMb, dimM),
                 dimNb, dimPb, dimMb, dimP, dimM, dimM);
    }
  }
}

/* Remaining code is the same as for the first version */
```

图 4-18 修订的串行矩阵乘法（没有给出与图 4-17 中的程序相同的部分）

通过变换，我们得到了一个组合了高级串行结构（关于 kb 的循环）和循环结构（关于 ib 和 jb 的嵌套循环）的程序，其中对于内部的嵌套循环，可以使用几何分解模式将其并行化。

**OpenMP 解决方案。**可以为一个共享内存环境生成这个程序的一个并行版本，方式是利用 OpenMP 的循环指令并行化内部的嵌套循环（关于 ib 和 / 或 jb）。像网格示例一样，保持较小的线程管理开销非常重要，因此 parallel 指令应当出现在关于 kb 的循环之外。应当将 for 指令放置在一个内部循环的前面。这个算法所带来的问题和最终的源代码修改实质上与网格程序示例一致，因此这里没有给出程序的源代码。

87 ~ 94

**MPI 解决方案。**图 4-19 和图 4-20 中给出了使用 MPI 的矩阵乘法程序的一个并行版本。采用 MPI 的自然方法是使用 SPMD 模式和几何分解模式。我们将使用先前描述的矩阵乘法算法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <mpi.h>
#define N 100

#define blockstart(M,i,j,rows_per_blk,cols_per_blk,stride) \
  (M + ((i)*(rows_per_blk))*(stride) + (j)*(cols_per_blk))

int main(int argc, char *argv[]) {
  /* matrix dimensions */
  int dimN = N; int dimP = N; int dimM = N;

  /* block dimensions */
  int dimNb, dimPb, dimMb;

  /* matrices */
  double *A, *B, *C;

  /* buffers for receiving sections of A, B from other processes */
  double *Abuffer, *Bbuffer;

  int numProcs, myID, myID_i, myID_j, NB;
  MPI_Status status;

  /* MPI initialization */
  MPI_Init(&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myID);

  /* initialize other variables */
  NB = (int) sqrt((double) numProcs);
  myID_i = myID / NB;
  myID_j = myID % NB;
  dimNb = dimN/NB; dimPb = dimP/NB; dimMb = dimM/NB;
  A = malloc(dimNb*dimPb*sizeof(double));
  B = malloc(dimPb*dimMb*sizeof(double));
  C = malloc(dimNb*dimMb*sizeof(double));
  Abuffer = malloc(dimNb*dimPb*sizeof(double));
  Bbuffer = malloc(dimPb*dimMb*sizeof(double));

  /* Initialize matrices */
  initialize(A, B, dimNb, dimPb, dimMb, NB, myID_i, myID_j);

/* continued in next figure */
```

图 4-19　使用消息传递的并行矩阵乘法（续见图 4-20）

```
/* continued from previous figure */

  /* Do the multiply */

  matclear(C, dimNb, dimMb, dimMb);
  for (int kb=0; kb < NB; ++kb) {

    if (myID_j == kb) {
      /* send A to other processes in the same "row" */
      for (int jb=0; jb < NB; ++jb) {
        if (jb != myID_j)
          MPI_Send(A, dimNb*dimPb, MPI_DOUBLE,
                   myID_i*NB + jb, 0, MPI_COMM_WORLD);
      }
      /* copy A to Abuffer */
      memcpy(Abuffer, A, dimNb*dimPb*sizeof(double));
    }
    else {
      MPI_Recv(Abuffer, dimNb*dimPb, MPI_DOUBLE,
               myID_i*NB + kb, 0, MPI_COMM_WORLD, &status);
    }
    if (myID_i == kb) {
      /* send B to other processes in the same "column" */
      for (int ib=0; ib < NB; ++ib) {
        if (ib != myID_i)
          MPI_Send(B, dimPb*dimMb, MPI_DOUBLE,
                   ib*NB + myID_j, 0, MPI_COMM_WORLD);
      }
      /* copy B to Bbuffer */
      memcpy(Bbuffer, B, dimPb*dimMb*sizeof(double));
    }
    else {
      MPI_Recv(Bbuffer, dimPb*dimMb, MPI_DOUBLE,
               kb*NB + myID_j, 0, MPI_COMM_WORLD, &status);
    }

    /* compute product of block[ib][kb] of A and
       block[kb][jb] of B and add to block[ib][jb] of C */
    matmul_add(Abuffer, Bbuffer, C,
               dimNb, dimPb, dimMb, dimPb, dimMb, dimMb);
  }
  /* Code to print results not shown */

  /* Clean up and end */
  MPI_Finalize();
  return 0;
}
```

图 4-20 使用消息传递的并行矩阵乘法（续图 4-19）

3 个矩阵（A、B 和 C）被分解为多个块。计算中所需要的 UE（在 MPI 中是指进程）被组织为一个网格，这样矩阵块的索引可以映射到进程的坐标上（即矩阵块（i,j）与第 i 行、第 j 列的进程相联系）。为简化起见，假设进程的数目 numProcs 是一个非常完美的方形，它的平方能够被矩阵的秩（N）整除。

尽管该算法一开始看可能比较复杂，但整体思想是简单易懂的。整个计算过程包含多个阶段（关于 kb 的循环）。在每一个阶段，行索引号等于 kb 的进程将它的 A 块发送到该行的所有进程上。同样地，索引号等于 kb 的进程将它的 B 块发送到该列的所有进程上。通信操作完成后，每一个进程将它所接收到的 A 块和 B 块相乘，并将结果累加到 C 块上。NB 个阶段之后，每一个进程上的 C 矩阵块组成最终的结果。

当使用 MPI 时，这些类型的算法非常常见。理解这些算法的关键是从以下几个方面进行考虑：进程的集合、每个进程所拥有的数据，以及随着计算的展开数据如何在相邻进程间流动。5.4 节、5.10 节和附录 B 将重新回顾这些问题。

人们已经对并行矩阵乘法和相关的线性代数算法开展了大量的研究。在 [FJL+88] 上给出了一种更复杂的方法，在该方法中，A 块和 B 块在进程间传播，仅当某个进程需要该块时，才到达该进程。

**知名应用。**大多数涉及微分方程的解的问题都使用几何分解模式。采用有限差分法直接映射到这个模式。使用这种模式的另一类问题来源于计算线性代数。ScaLAPACK [Sca, BCC+97] 库中的并行函数大多数基于这种模式。这两类问题涵盖了科学计算中的大部分并行应用程序。

6. 相关模式

如果每一个块的更新可以不使用其他块中的数据来完成，则这种模式简化为 4.4 节描述的易并行算法。作为这种计算的一个示例，考虑一个 2D FFT（快速傅里叶变换）的计算，首先对矩阵的每一行应用一次 1D FFT，再对矩阵的每一列应用一次 1D FFT。尽管这个分解看上去好像是基于数据的（基于行或基于列），但事实上，计算由*任务并行*模式的两个实例组成。

如果待分配的数据结构本质上是递归的，则可以应用*分治*模式或*递归数据*模式。

## 4.7 递归数据模式

1. 问题

假设问题涉及对递归数据结构的操作（如列表、树或图）的操作，这些操作看上去好像需要串行处理。如何能够对这些数据结构并行地执行相关操作？

2. 背景

某些问题具有递归数据结构，很自然地使用 4.5 节描述的分治策略来利用它们固有的潜在并发性。但是，对这些数据结构的其他操作看上去似乎具有很少的潜在并发性，因为表面上求解这些问题的仅有方式是：沿着数据结构移动，在移动到下一个元素之前，计算当前元素的结果。但是，有时候可以重塑操作，使得程序能够对数据结构的所有元素并发地操作。 97

[J92] 中的一个例子演示了这种情形：假设具有一个有根有向树（定义方式为：对于每一个节点，都有它的直接祖先，而根节点的祖先为它自己）的森林，并且对于森林中的每一个节点，我们想计算包含该节点的树的根。在一个串行程序中，为了完成该任务，将使用深度优先法搜索每一棵树，从树的根节点搜索到它的叶子节点。当访问每一个节点时，我们具有对应根节点的信息。对于一个具有 $N$ 个节点的森林，程序的总运行时间是 $O(N)$。虽然存在一些潜在的并发性（对子树进行并发操作），但没有一种显而易见的方法对所有元素进行并发操作，因为我们无法在获得某个特定节点父节点根的情况下，获知该特定节点的根。

但是，重新思考该问题将发现其他的并发性：我们首先为每一个节点定义一个"后继"，初始时后继是该节点的父节点，最终将是该节点所属的树的根节点。然后我们计算每一个节点的"后继的后继"。对于到根节点只有一个"跃距"的节点，计算不改变该节点后继的值（因为根节点的父节点是它自己）。对于到根节点至少有两个"跃距"的节点，计算使得该节

点的后继成为它的父节点的父节点。我们重复这个计算，直到它收敛（即某一步所产生的值与它前面一步所产生的值相同）。在该点处，每一个节点的后继都是目标值。图 4-21 演示了需要 3 步获得收敛的示例。在每一步，我们可以并发地对树中所有的 $N$ 个节点进行操作，并且该算法最多在 log$N$ 步获得收敛。

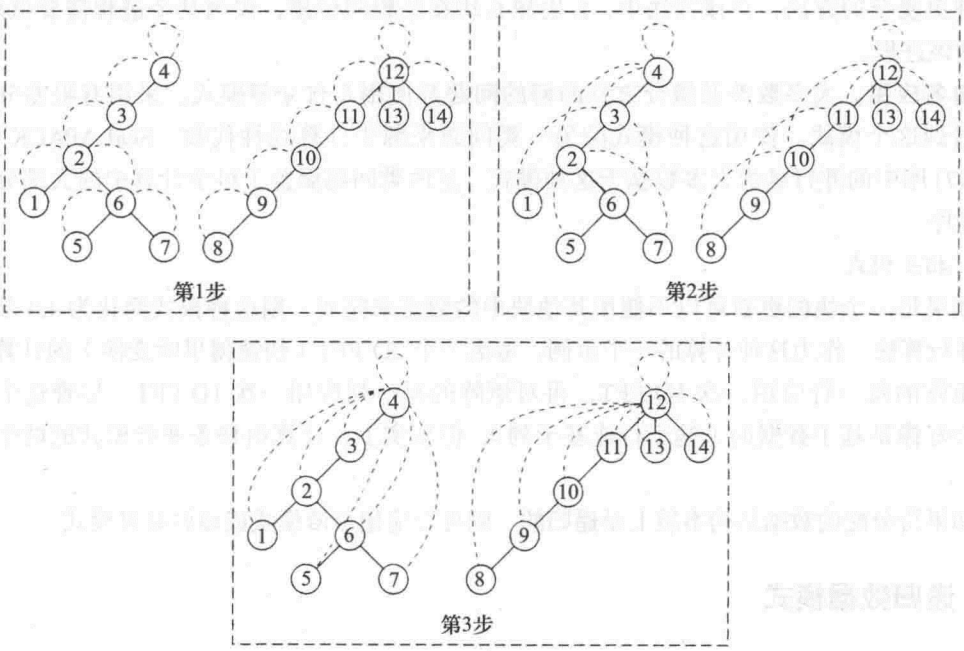



图 4-21 在一个森林中寻找根节点（实线表示节点间的初始父子关系，虚线表示从节点指向其后继）

我们所做的是将初始串行计算（为距离跟节点一个“跃距”的节点找到根，然后为距离根节点两个“跃距”的节点找到根，等等）转换为另外一个计算，该计算为每一个节点计算部分结果（后继），然后重复地组合这些部分结果，首先组合相邻结果，然后组合距离两个跃距的节点的部分结果，然后组合距离 4 个跃距的节点的部分结果，以此类推。这种策略可以应用到其他乍一看好像必须使用串行方法的问题中，在本节的示例部分中提供了其他一些示例。这种技术有时称为*指针跳转*（pointer jumping）或者*双递归*（recursive doubling）。

这种新构造的算法一个令人感兴趣的方面是，新的算法包含的工作比初始串行算法多得多——$O(N\log N)$ 比 $O(N)$，但新构造的算法包含潜在的并发性，如果彻底地挖掘，则总运行时间将减少为 $O(\log N)$（而初始算法为 $O(N)$）。基于这种模式的大多数策略和算法都采取了相似的折衷策略，即以总工作量的增加来降低总执行时间的减少。还要注意的是，可挖掘的并发性可以非常细粒度（如前面的示例所示），这可能会限制该模式产生一个高效的算法。虽然如此，但是这种模式仍然可以作为一个启发，用于考虑对那些第一眼看上去似乎必须使用串行方法求解的问题的并行化。

3. *面临的问题*

- 重新构造问题，以便将递归数据结构的一种固有串行遍历转换为另一种允许所有元素并行操作的遍历，这样做将增加计算的总工作量。必须平衡这种情况，以提高并发运行时的性能。

- 这种重新构造可能会很困难（因为这需要从一个不寻常的角度观察初始问题），并且可能会得到一个难于理解和维护的设计。
- 这种模式所提供的并发性是否能高效地挖掘以提高性能，依赖于在目标并行计算机系统中，操作的计算开销和与计算相关的通信开销。

4. 解决方案

应用这种模式最具有挑战性的部分是，将对一个递归数据结构的操作重新构造为一种能够提供额外的并发性的形式。一般的指导原则难以完成此类构造，但是根据这种模式提供的示例，其关键思想应当是清晰的。

在并发性已经揭示后，这种并发性不一定总是能够高效地挖掘，并加速程序的求解速度。这依赖于多种因素，主要包括更新递归数据结构的每一个元素所包含的工作量和目标并行计算机的特征。

98 ~ 99

**数据分解**。在这种模式中，递归数据结构被彻底地分解为独立的元素，每一个元素被分配给一个独立的 UE。理想情况下，每一个 UE 将被分配给一个不同的 PE，但将多个 UE 分配给一个 PE 是也可能的。但是，如果每一个 PE 上 UE 的数目太大，则整体性能会较差，因为没有足够的并发性去抵消总的工作量的增加。

例如，考虑前面描述的根查找的问题。我们将忽略计算中的开销。如果 $N$=1024，$t$ 是对一个数据元素执行一步的时间，则串行算法的运行时间大约是 1024$t$。如果每个 UE 被分配给它自己的 PE，则并行算法的运行时间约为 (log$N$)$t$ 或 10$t$。但是，如果并行算法只能利用两个 PE，则每个元素的 $N$log$N$ 或 1024 步计算必须在两个 PE 上执行，则执行时间至少为 5120$t$，远大于串行算法的时间。

**结构**。应用这种模式的典型结果是一个算法，该算法的高层结构是一个循环形式的串行组合，其中循环的每一次迭代可以描述为“对递归数据结构的所有（或选择的）元素同时地执行这种操作”。典型操作包括“使用每一个元素的后继的后继替换它的后继”（参考本节前面的示例）和“使用当前元素的值与前趋元素的值来替换当前元素的值”。

**同步**。符合这种模式的算法被描述为同时更新数据结构的所有元素。这些目标平台（例如，以早期 Connection Machine 机器为例的 SIMD 体系结构）通过将每一个数据元素分配给一个独立的 PE（可能是一个逻辑 PE），并在每一个 PE 上以一种步调一致的模式执行指令，来完成该工作。支持编程环境（例如，高性能的 Fortran [HPF97]）的 MIMD 平台也能提供相似语义。

如果目标平台不隐式提供所需要同步，则用户需要显式地引入同步。例如，如果在一个循环迭代期间执行的操作包含如下赋值语句：

```
next[k] = next[next[k]]
```

则并行算法必须确保在其他需要使用 `next[k]` 的值用于计算的 UE 接收到该值之前，`next[k]` 未更新。一种常用的技术是引入一个新变量 `next2`。然后偶数次迭代读取 `next` 值，但更新 `next2`，而奇数次读取 `next2` 的值，并更新 `next`。通过在每一对连续的迭代之间放置一个栅栏（如第 6 章所描述的一样），来完成所必需的同步。注意，这实际上增加了与并行算法相关的额外开销，它可能会抵消额外的并发性所带来的加速比。如果每一个元素所需要的计算都很少（许多示例都是这样的），这很可能成为一个因素。

100

如果 PE 的数目比数据元素少，则程序设计者必须确定是否将每一个数据元素分配给一个 UE，并将多个 UE 分配给每一个 PE（从而仿真这些并行性），或者是否将多个数据元素分配给每一个 UE，然后串行地执行它们。后者较复杂（需要一个与前面描述的相似方法，其中同步更新所涉及的变量被复制），但效率较高。

5. 示例

**链表的部分和**。这个示例源于 Hilis 和 Steele[HS86]，问题是计算一个链表中所有元素的前缀和（prefix sum），其中链表中的每一个元素包含一个值 $x$。换句话说，在计算完成后，第一个元素将包含 $x_0$，第二个元素将包含 $x_0+x_1$，第三个元素将包含 $x_0+x_1+x_2$，依次类推。

图 4-22 给出了这个基本算法的伪代码。图 4-23 给出了这个计算的演化过程，其中 $x_i$ 是列表中第（$i$+1）个元素的初始值。

```
for all k in parallel
{
    temp[k] = next[k];
    while temp[k] != null
    {
        x[temp[k]] = x[k] + x[temp[k]];
        temp[k] = temp[temp[k]];
    }
}
```

图 4-22 计算链表的部分和的伪码

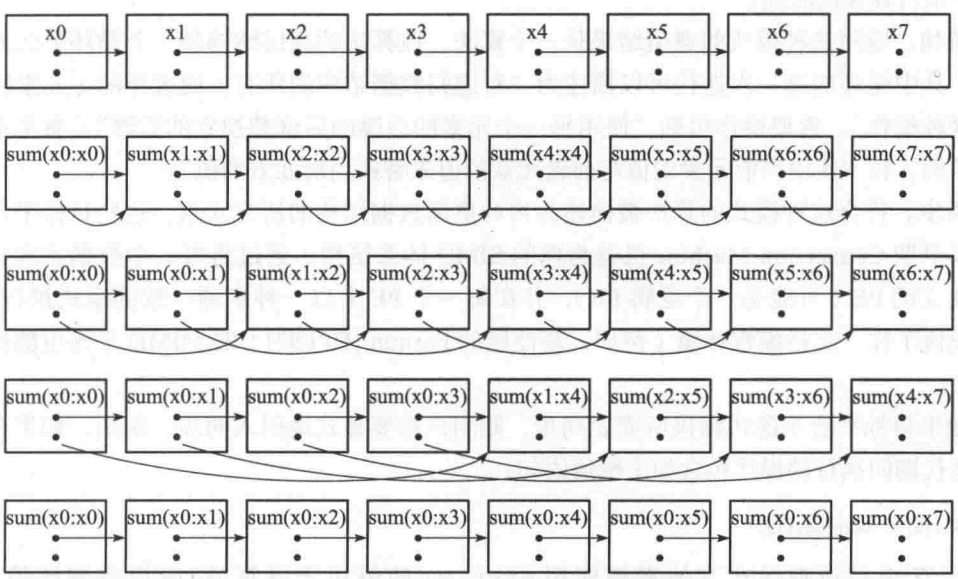



图 4-23 计算链表的部分和的步骤（直线箭头表示元素间的链接，曲线箭头表示加法）

通过将加法替换为相关运算符，可以将这个示例进行归纳，该示例有时也称为*前缀扫描*。它可以被使用在多种情形中，包括求解各种类型的递推关系。

**知名应用**。基于这种模式开发的算法是一种*数据并行*算法。它们广泛地应用在 SPMD 平台中，是对高性能 Fortran [HPF97] 等语言的一种程度较小的扩展。这些平台支持这种模式所

需要的细粒度并发性，并且自动处理同步，因为在所有的处理器上每一个计算步骤（物理上的或逻辑上的）都是步调一致地发生的。Hillis 和 Steele[HS86] 描述了几种关于这种模式的有趣应用，包括寻找一个链表的尾部，计算一个链表的所有部分和，二维图像中的区域标注和分析。

在组合优化中，遍历一个图或树中所有节点的问题也可以利用这种模式来求解，方法是先通过寻找关于节点的一种排序来创建一个列表。欧拉环游和耳朵分解（ear decomposition）[EG88] 是计算这种排序的著名技术。

JáJá [J92] 也描述了这种模式的几种应用：寻找一个由有根有向树组成的森林中树的根，计算一个有根有向树中集合的部分和（与前面链表的示例相似），以及计算列表的范围（确定列表的每一个元素与列表的起始点 / 终点的距离）。

6. 相关模式

就实际的并发性而言，这种模式与几何分解模式非常相似，不同点在于这种模式中的数据结构是递归的（至少在概念上是这样的），并且这些元素可以同时操作。这种模式的不同点在于强调重新考虑问题来挖掘细粒度并行性。

101 ~ 102

## 4.8 流水线模式

1. 问题

假设整个计算涉及对很多数据集进行计算，其中计算可看作是对通过一个步骤序列的数据流的处理。如何挖掘这种潜在的并发性呢？

2. 背景

流水线与这种模式非常类似。假设我们想制造大量的汽车。制造过程可以分解为一个操作序列，其中的每一步操作都为汽车添加某个部件，例如，发动机或者挡风玻璃等。流水线为每一个工人分配一个部件。当每一辆汽车在流水线中移动时，每一个工人对连续流动的汽车安装同样的部件。在流水线充满之后（并且直到它开始清空为止），所有工人可以同时保持繁忙，所有人都对其位置处的汽车并发地执行他们的操作。

可以在计算机系统的多个粒度层次发现流水线的示例，包括 CPU 硬件本身。

- **现代 CPU 中的指令流水线。**指令执行的所有阶段（取指、解码、执行等）以流水线方式完成；当解码一条指令时，它的前驱正在执行指令，它的后继正在被取指令。
- **向量处理（循环级流水线）。**在某些超级计算机中，专门的硬件使得对向量的操作能够以流水线的模式完成。典型情况下，编译器能够识别出循环，例如：

```
for(i = 0; i < N; i++) { a[i] = b[i] + c[i]; }
```

  该循环可以向量化，且专门的硬件能够实现它。在一个短暂的启动时间之后，每一个时钟周期将产生一个 `a[i]` 值。
- **算法级流水线。**许多算法可以表示为递归关系，并使用流水线或者较高维概括（脉动阵列）来实现。考虑性能原因，这通常需要开发专门的硬件来实现。
- **信号处理。**实时传感器数据流通过一个过滤器序列可以被模拟为一个流水线，其中每一个过滤器对应于流水线的一个阶段。

- **图。**通过对图像序列中的每一个图像应用相同的操作序列图像，序列的处理可以被模
103 型化为一个流水线，其中每一个操作对应于一个流水线阶段。某些阶段可以使用专门的硬件实现。
- **UNIX 中的 shell 程序。**例如，shell 命令

```
cat sampleFile | grep "word" | wc
```

创建了一个三阶段的流水线，其中每个命令（cat、grep 和 wc）对应一个进程。

这些示例和工业流水线通常具有几个方面的共同点。对一个关于数据元素序列（在工业流水线中，指的是汽车）中的每一个元素都应用一个操作序列去处理（在工业流水线情形中，操作是指安装发动机、挡风玻璃等）。尽管对单个数据元素的操作可能有顺序的约束（例如，在安装引擎罩之前，需要先安装发电机），但是可以同时对不同的数据元素执行不同的操作（例如，可以对一辆汽车安装发电机，而同时对另一辆汽车安装引擎罩）。

同时对不同的数据元素执行不同操作的能力是这种模式所挖掘的潜在并发性。根据第 3 章描述的内容，每个任务由对一个数据元素（类似于一个工业流水线工人安装一个部件）重复地应用一个操作组成，任务间的依赖性是强制某种顺序的顺序约束，必须采用该顺序对每一个数据元素执行操作（比如，在安装引擎罩之前，必须先安装好引擎）。

3. 面临的问题

- 一个优秀的解决方法应当尽可能简单地表示顺序约束。在问题中，顺序约束是简单并规范的，且能够通过流过流水线的数据流来表示。
- 目标平台包含某些专用硬件，这些硬件执行一些所期望的操作。
- 在某些应用中，期望将来能够对流水线的阶段进行添加、修改或重新排序。
- 在某些应用中，输入序列中的临时条目可以包含一些错误来防止元素的处理。

4. 解决方案

104 这种模式的关键思想源于工业流水线：通过将每一个操作（流水线的阶段）分配给一个不同的工人，并且让他们同时工作，随着元素操作完成，数据元素从一个工人处传递到下一个工人处，从而挖掘潜在的并发性。在并行编程方面，思想是：将每一个任务（流水线的阶段）分配给一个 UE，并提供一种机制，使得流水线中的每个阶段都可以将数据元素发送到下一个阶段。这种策略可能是处理这种类型的顺序约束的最简单方式。通过将流水线的阶段合理地映射到 PE 上，该策略使得应用程序能够充分利用专用硬件的优点，并提供合理的错误处理机制，后面将描述这一点。该策略也可能产生一种模块化设计，可以在以后扩展或修改。

在进一步介绍之前，演示流水线应该如何操作是有益的。$C_i$ 表示对数据元素 $i$ 的多步操作。$C_i(j)$ 是该计算的第 $j$ 步。将计算步骤映射到流水线的阶段，使得流水线的每一个阶段计算一个步骤。初始时，流水线的第一个阶段执行 $C_1(1)$。完成该步后，流水线的第二个阶段接收第一个数据项并计算 $C_1(2)$，而流水线的第一阶段计算第二个数据项的第一步 $C_2(1)$。接着，流水线的第三个阶段计算 $C_1(3)$，而第二个阶段计算 $C_2(2)$，第一个阶段计算 $C_3(1)$。图 4-24 演示了由四个阶段组成的流水线是如何工作的。注意，初始时并发性是受限的，并且某些资源处于闲置状态，直到有用的工作占满流水线的各个阶段。这称为*流水线的填充*。在计算的末尾（*流水线的排空*），当最后一项元素在流水线中处理时，又存在一个受限的并发性和一些空

闲的资源。我们希望花费在填充和排空流水线上的时间远小于计算的总时间。如果流水线的阶段数目小于所要处理的元素的数目，则将满足我们所期望的情形。另外，如果流水线每一个阶段处理一个数据元素所花费的时间近似相等，则整体吞吐量 / 效率将最大化。

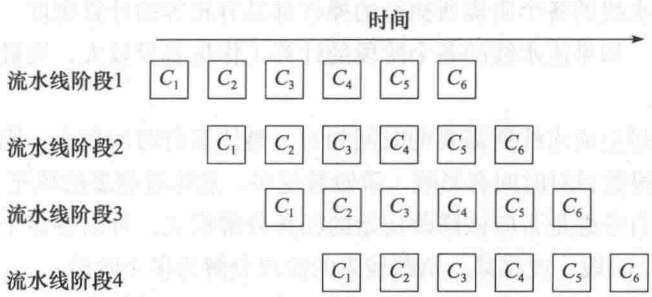



图 4-24 一个流水线的操作（流水线的每个阶段 $i$ 完成第 $i$ 个计算步骤）

这种思想可以扩展到比完全线性的流水线更通用的情形。例如，图 4-25 演示了两种流水线，每一种流水线包含 4 个阶段。在第二个流水线中，第三个阶段包含两种操作，这两种操作可以并发地执行。 105

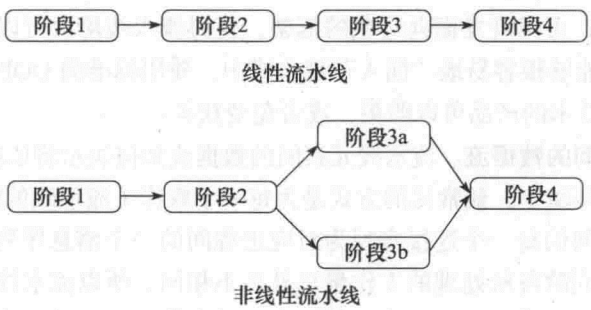



图 4-25 流水线示例

**定义流水线阶段**。通常每一个流水线阶段对应一个任务。图 4-26 给出了每个阶段的基本结构。

```
initialize
while (more data)
{
    receive data element from previous stage
    perform operation on data element
    send data element to next stage
}
finalize
```

图 4-26 流水线阶段的基本结构

如果将要处理的数据元素的数目能够提前知道，则每一个阶段能够统计出元素的数目，并且在处理完这些元素后，该阶段终止。另外，可以在流水线中发送一个表示终止的标识。

此时需要考虑影响性能的几个因素。

- 在整个流水线中，并发量受限于阶段的数目。这样，阶段的数目越多将带来更多的并发性。但是，数据序列必须在阶段之间传递，这给计算引入了额外的开销。这样，我

们需要将计算组织为几个阶段，每一个阶段所完成的工作量比通信开销大。工作量的大小高度依赖于特定的体系结构。专用硬件（如向量处理器）允许非常细粒度的并行性。

- 如果通过流水线的各个阶段所执行的操作都具有相等的计算密度，则这种模式将能够工作得更好。如果流水线的各个阶段的计算工作量差异较大，则最慢的阶段将是总吞
106 吐量的瓶颈。
- 如果填充和排空流水线所需要的时间相对于整体运行时间较小，则这种模式能够工作得更好。阶段数目对时间有影响（阶段数越多，意味着越多的填充 / 排空时间）。

因此，此时应当考虑是否应该修改初始的任务分解模式，可以将多个负载较轻的相邻流水线阶段组合为一个阶段，或将某个负载较重的阶段分解为多个阶段。

也值得考虑使用其他*算法结构*模式将一个负载较重的阶段并行化。例如，如果流水线正在处理图像序列，通常可以采用*任务并行*模式将每一个阶段并行化。

**构造计算。**我们也需要一种方法来构造整体的计算。一种可行的方式是使用 SPMD 模式（见第 5 章），并使用每一条 UE 的 ID 来选择一条 case 或 switch 语句中的某一项，其中每个 case 对应于流水线的一个阶段。

为了提高模块性，可以开发面向对象的框架，使得流水线阶段可以使用对象或过程来表达，这些对象或过程能够很容易地“插入”流水线中。使用标准的 OOP 技术构造这样的框架并不困难，其中几种技术的产品可以购买，或者免费获得。

**描述流水线元素间的数据流。**流水线元素间的数据流如何表示将依赖于目标平台。

在一个消息传递环境中，最常见的方式是为每一个操作（流水线的阶段）分配一个进程，并将流水线连续阶段间的每一个连接实现为对应进程间的一个消息序列。因为各个阶段很难很好地同步，并且不同阶段所处理的工作量总是各不相同，所以流水线阶段间的数据流通常必须缓冲和排序。大多数消息传递环境（如 MPI）很容易实现这种需求。如果发送个别消息的开销较高，则应当考虑在每条消息中发送多个数据元素，该方法以增加填充流水线时间为代价，降低了总通信开销。

如果一个消息传递编程环境不能很好地适合目标平台，则可以显式地使用缓冲通道连接流水线的各个阶段。该缓冲通道可以由发送任务和接收任务之间所共享的队列来实现，该队列使用*共享队列*模式。

如果流水线的各个阶段分别实现为并行程序，则可能需要更复杂的方法，特别是如果需要在各阶段之间执行某种形式的数据再分配。例如，如果数据需要在不同的维中分解，或者在相同维中分解为不同数目的子集，则将出现这种情形。例如，一个应用程序的某一个阶段
107 中的每一个数据元素被分解为 3 个子集，而另一个阶段中的每一个数据元素被分解为 4 个子集。处理这种情形的最简单方式是聚合和分解阶段间的数据元素。一种方法是在每一个阶段只有一个任务与其他阶段的任务通信，然后这个任务将在它的阶段中负责与其他任务的交互，以分配输入数据元素和收集输出数据元素。另外一种方法是引入额外的流水线阶段来执行聚合或分解操作。然而，这些方法都涉及大量的通信。最好是在较早的阶段中能够明确它的后继的需求，并与每一个接收它的部分数据的任务直接通信，而不是在一个阶段聚合数据，然后在另外一个阶段分解数据。这种方法以降低简单性、模块性和灵活性为代价，来提高性能。

与传统方法不同，运行在工作站集群上的网络文件系统已经用于流水线阶段间的通信。

数据被某个阶段写到一个文件中，并被它的后继阶段读取。网络文件系统通常都比较成熟，且经过很好的优化，还提供在所有的 PE 上的文件的可视化和并发控制机制。运行在网络文件系统之上的较高层抽象也可以使用，例如，元组空间和黑板。基于文件系统的解决方案适用于粗粒度的应用程序，其中处理数据所需要的时间比访问文件系统所需要的时间多得多。

**错误处理**。对于某些应用程序，可能需要妥善地处理错误情况。一种解决方案是创建一个独立的任务来处理错误。常规流水线的每一个阶段将它不能处理的任何数据元素和相关错误信息发送给这个任务，然后继续处理流水线中的下一项。负责处理错误的任务恰当地处理这些错误数据元素。

**处理器分配和任务调度**。最简单的方式是为流水线的每一个阶段分配一个 PE。如果 PE 相似，并且对于每一个阶段来说，处理一个数据元素所需要的工作量大致相等，则将获得很好的负载平衡。如果各个阶段需求不同（例如，某个阶段需要在专用硬件上运行），这种情况应当在给流水线阶段分配 PE 时考虑。

如果 PE 的数目比流水线的阶段数目少，则必须将多个阶段分配给同一个 PE，最好采用一种能够提高或者至少不降低系统整体性能的方式。不需要共享资源的多个阶段可以分配给同一个 PE，例如，一个写入硬盘的阶段和一个主要涉及 CPU 计算的阶段比较适合共享一个 PE。如果处理一个数据元素所需要的工作量在各阶段间不同，则包含较少工作量的几个阶段可以分配给同一个 PE，从而改善了负载平衡。将相邻阶段分配给同一个 PE 可以降低通信开销。因此值得考虑将流水线的相邻阶段组合为单个阶段。

如果 PE 数目比流水线的阶段数目多，则应该考虑使用一种恰当的*算法结构*模式将一个 108
或多个流水线阶段并行化，像前面讨论的一样，可以将多个 PE 分配给并行化的阶段。如果并行化的阶段起初是系统的一个瓶颈（需要花费比其他阶段更多的时间而降低了整体性能），则这种方法非常有效。

另外，需要重新利用比流水线阶段数更多的 PE，如果数据项之间没有时间约束（也就是说，数据项 3 可以在数据项 2 之前计算），则可以并行地运行多个独立的流水线。可以认为这是*任务并行*模式的一个实例。该方法将提高整体计算的吞吐量，但并没有明显增加延迟，因为一个数据元素仍然需要花费相等的时间来穿越流水线。

**吞吐量和延迟**。当评估一个给定的设计是否会产生可接受的性能时，需要牢记几个因素。

在许多使用流水线模式的情形中，有趣的性能度量是吞吐量，即流水线充满之后，每一个时间单元内所能够处理的数据项数目。例如，如果流水线的输出是一系列渲染图像，则流水线必须具有充足的吞吐量（每个时间单元所能够处理的数据项数目），按要求的帧速率生成图像。

在另一种情形中，输入可能来源于实时采样的传感器数据。此时，对吞吐量（当数据到来时，流水线应当能够处理所有的数据，而不是将它们备份在输入队列中，也不能造成数据丢失）和延迟（从一个输入元素的产生，到处理完该输入所花费的总时间量）都具有限制。在这种情形中，需要最小化受限的延迟来确保具有足够的吞吐量，以处理输入数据。

5. 示例

**傅里叶变换计算**。信号处理中广泛使用的一种计算包括对不同的数据集重复地执行下列计算。

1）对数据集的每一个元素执行离散傅里叶变换（DFT）；

2）处理变换元素的结果；

3）对于所处理的结果执行逆 DFT。

这类计算的示例包括卷积、相关性和过滤操作 [PTV93]。

这种形式的计算可以利用一个 3 阶段流水线方便地完成。

- 流水线的第一个阶段执行初始的傅里叶变换；它重复地获得一个输入数据集，执行变换操作，并将结果传递给流水线的第二个阶段。
- 流水线的第二个阶段执行所期望的元素处理；它重复地从流水线的第一个阶段获得部分结果（对输入数据集应用最初的傅里叶变换所得的结果），执行处理，并将结果传递给流水线的第三个阶段。通常可以使用*算法结构*模式中的其他模式将这个阶段本身并行化。
- 流水线的第三个阶段执行最后的逆傅里叶变换；它重复地从流水线的第二个阶段获得部分结果（对输入数据应用最初的傅里叶变换，然后应用元素处理后所得的结果），执行逆傅里叶变换，并输出结果。

109

流水线的每个阶段一次处理一个数据集。然而，除了流水线的开始填充期之外，流水线的所有阶段能够并发地操作。当第一个阶段正在处理第 $N$ 个数据集时，第二个阶段正在处理第（$N$–1）个数据集，而第三个阶段正在处理第（$N$–2）个数据集。

**Java 流水线框架。**这个例子的图形为流水线演示了一个简单的 Java 框架和一个示例应用程序。

这个框架由一个关于流水线阶段的基类（ipelinestage，如图 4-27 所示）和一个关于流水线的基类（linearPipeline），如图 4-28 所示）组成。应用程序为每一个阶段提供 PipelineStage 类的一个子类，实现 PipelineStage 类的 3 个抽象方法，来表示哪个阶段应当完成初始步骤，哪个阶段应当完成计算步骤，哪个阶段应当完成最后的步骤，并且提供 LinearPipeline 类的一个子类，实现 LinearPipeline 类的所有抽象方法，用以创建一个包含流水线阶段的数组和连接各个阶段的期望队列。对于连接各个阶段的队列，使用 LinkedBlockingQueue 类，它是 BlockingQueue 接口的一个实现。这些类位于 java.util.concurrent 包中。使用类名声明队列所包含的对象类型。例如，new LineKedBlockingQueue<String> 创建一个 BlockingQueue，它由能够存储 String 的基本链表实现。令人感兴趣的操作是 put（即向队列中添加一个对象）和 take（即移除一个对象）。如果队列为空，则 take 被阻塞。CountDownLatch 类也位于 java.util.concurrent 包中，该类是一个简单的栅栏，它允许程序在终止时输出一条消息。普通的栅栏和特殊的 CountDownLatch 将在第 6 章中讨论。

```
import java.util.concurrent.*;

abstract class PipelineStage implements Runnable {

    BlockingQueue in;
    BlockingQueue out;
    CountDownLatch s;

    boolean done;

    //override to specify initialization step
```

图 4-27 流水线阶段的基类

```
    abstract void firstStep() throws Exception;
    //override to specify compute step
    abstract void step() throws Exception;
    //override to specify finalization step
    abstract void lastStep() throws Exception;

    void handleComputeException(Exception e)
    { e.printStackTrace(); }

    public void run()
    {
     try
       { firstStep();
         while(!done){ step();}
         lastStep();
       }
     catch(Exception e){handleComputeException(e);}
       finally {s.countDown();}
    }
    public void init(BlockingQueue in,
                     BlockingQueue out,
                     CountDownLatch s)
    { this.in = in; this.out = out; this.s = s;}
}
```

图 4-27 （续）

```
import java.util.concurrent.*;

abstract class LinearPipeline {
    PipelineStage[] stages;
    BlockingQueue[] queues;
    int numStages;
    CountDownLatch s;

    //override method to create desired array of pipeline stage objects
    abstract PipelineStage[] getPipelineStages(String[] args);

    //override method to create desired array of BlockingQueues
    //element i of returned array contains queue between stages i and i+1
    abstract BlockingQueue[] getQueues(String[] args);

    LinearPipeline(String[] args)
    { stages = getPipelineStages(args);
      queues = getQueues(args);
      numStages = stages.length;
      s = new CountDownLatch(numStages);

      BlockingQueue in = null;
      BlockingQueue out = queues[0];
      for (int i = 0; i != numStages; i++)
      { stages[i].init(in,out,s);
        in = out;
        if (i < numStages-2) out = queues[i+1]; else out = null;
        }
     }

     public void start()
     { for (int i = 0; i != numStages; i++)
        { new Thread(stages[i]).start();
        }
     }
}
```

图 4-28 线性流水线的基类

其余的图展示了一个示例应用程序（整数排序的流水线）的代码。图 4-29 是 Linearpipeline 需要的子类，图 4-30 是 Pipeline Stage 必需的子类。这里未显示生成输入或读取输入，及处理输入的附加流水线阶段。

```
import java.util.concurrent.*;

class SortingPipeline extends LinearPipeline {

  /*Creates an array of pipeline stages with the
  number of sorting stages given via args. Input
  and output stages are also included at the
  beginning and end of the array. Details are omitted.
  */
   PipelineStage[] getPipelineStages(String[] args)
    { //....
       return stages;
    }


    /* Creates an array of LinkedBlockingQueues to serve as
        communication channels between the stages. For this
        example, the first is restricted to hold Strings,
        the rest can hold Comparables. */
    BlockingQueue[] getQueues(String[] args)
    { BlockingQueue[] queues = new BlockingQueue[totalStages - 1];
       queues[0] = new LinkedBlockingQueue<String>();
       for (int i = 1; i!= totalStages -1; i++)
       { queues[i] = new LinkedBlockingQueue<Comparable>();}
       return queues;
    }


    SortingPipeline(String[] args)
    { super(args);
   }


    public static void main(String[] args)
      throws InterruptedException
    { //create pipeline
       LinearPipeline l = new SortingPipeline(args);
       l.start(); //start threads associated with stages
       l.s.await(); //terminate thread when all stages terminated.
       System.out.println("All threads terminated");
    }
}
```

图 4-29 流水线排序（主类）

```
class SortingStage extends PipelineStage
{
    Comparable val = null;
    Comparable input = null;

    void firstStep() throws InterruptedException
    { input = (Comparable)in.take();
       done = (input.equals("DONE"));
       val = input;
       return;
    }
```

图 4-30 流水线排序（排序阶段）

```
    void step() throws InterruptedException
    { input = (Comparable)in.take();
      done = (input.equals("DONE"));
      if (!done)
      { if(val.compareTo(input)<0)
        { out.put(val); val = input; }
        else { out.put(input); }
      } else out.put(val);
    }

    void lastStep() throws InterruptedException
    { out.put("DONE"); }
}
```

图 4-30 （续）

**知名应用**。信号处理和图像处理的许多应用都可以通过流水线实现。

OPUS [SR98] 系统是一个由太空望远镜科学协会开发的流水线框架，起初用于处理哈勃空间望远镜测得的遥测数据，后来还应用于其他应用程序。OPUS 使用一种基于网络文件系统作为阶段间通信的黑板体系结构。OPUS 还包含一些监控工具，并支持错误处理。

机载监视雷达使用了时空自适应处理（STAP）算法，该算法被实现为一个并行流水线 [CLW+00]。每个阶段本身就是一个并行算法，流水线需要数据在某些阶段间重新分配。

Fx[GoS94] 是一个基于 HPF [HPF97] 的并行 Fortran 编译器，已经用于开发多个示例应用程序 [DGO+94、SSOG93]，这些应用结合了数据并行性（类似于 4.6 节提到的并行形式）和流水线。例如，某个应用通过一个两阶段的流水线，（其中，一个阶段用于行变换，另一个阶段用于列变换），其中每一个阶段本身利用数据的并行性完成并行化。论文 SIGPLAN（[SSOG93]）中特别有趣的是通过性能数据完成对基于数据的并行方法和该方法的比较。

[J92] 提供了一些关于流水线的较细粒度的应用，包括向一棵树中插入元素序列和基于流水线的归并排序。

6. 相关模式

这种模式非常类似于 [BMR+96] 中介绍的管道与过滤器模式，主要差别是该模式显式地论述了并发性。

对于那些输入数据间没有时间依赖性的应用程序，使用这种模式的另一种方式是，利用任务并行模式设计一种基于多个串行流水线的应用，这些流水线并行执行。

乍一看，我们也可据此推测：使用责任链模式 [CHJV95]（Chain of Responsibility pattern）构建的串行解决方案可以很容易地使用流水线模式完成并行化。在责任链（COR）中，一个“事件”沿着对象链传递，直到一个或多个对象处理该事件。某些环境直接支持这种模式，例如，在 Java Servlet 规范 [SER] ⊖中使用这种模式进行 HTTP 请求的过滤。但是对于 Servlet 和其他一些典型的 COR 应用程序来说，使用这种模式的原因是为了支持一个程序的模块化构建，其中，程序需要根据事件的类型而选择不同的方式来处理独立的事件。可能在链中只有一个对象处理该事件。在大多数情形中，我们期望任务并行模式比流水线模式更加适合。事实上，Servlet 容器实现已经支持多线程来处理独立的 HTTP 请求，且免费提供该解决方法。

⊖ Servlet 是 Web 服务器调用的 Java 程序。Java Servlet 技术包含在针对 Web 服务器应用程序的 Java 2 Enterprise Edition 平台中。

流水线模式与基于事件的协作模式类似，两种模式都应用于那些很自然地将计算分解为半独立任务集合的问题。其中，不同点是基于事件的协作模式是不规则的和异步的，而流水线模式是规则的和同步的。在流水线模式中，半独立任务表示流水线的各个阶段，流水线的结构是静态的，连续的阶段之间的交互是规则的并且是松散同步的。然而，在基于事件的协作模式中，任务之间以非常不规则和异步的方式进行交互，并且没有对静态结构的需求。

## 4.9 基于事件的协作模式

1. 问题

110 ~ 114

假设可以把问题分解为半独立任务的分组，并且这些任务以一种非规范的模式交互。任务之间的数据流交互隐含了任务间的顺序约束。如何实现这些任务和任务间的交互，使得它们能够并发地执行？

2. 背景

一些问题可以非常自然地表示为一个关于半独立实体的集合，这些半独立实体以一种不规则的方式交互。如果对比这种模式和流水线模式，这就是最清晰的差别。在流水线模式中，实体组成一个线性流水线，每一个实体仅与该实体两侧的实体交互，数据流是单向的，交互在相当规范和可预测的时间间隔中进行。正好相反，在基于事件的协作模式中，不存在对线性结构的限制，也不限制数据流必须是单向的，并且交互在不规则和不可预测的时间间隔中进行。

以现实世界的编辑部作为类比，编辑部具有记者、编辑、复审人员和其他收集报道题材的雇员。当记者完成报道时，他们将报道发送给对应的编辑；编辑确定将报道发送给复审者（他将最终报道返回）或者返回给记者做进一步修改。每一个雇员是一个半独立个体，他们的交互（例如，一个记者将一个报道发送给一个编辑）是不规范的。

在离散事件仿真领域可以发现许多其他示例。所谓的离散事件仿真，是对一个由对象集合组成的物理系统的仿真，对象间的交互通过离散“事件”序列表示。这类系统的一个示例是 [Mis86] 中描述的汽车清洗工厂，这个工厂有两台汽车清洗机器和一个服务员。汽车到达服务员处的时间是随机的，如果有一台机器空闲，服务员把汽车引向该汽车清洗机处，如果两台机器都繁忙，则排队等待。每一台汽车清洗机一次只清洗一辆汽车。对于一个给定的分布或者到达时间，目标是计算一辆汽车花费在该系统中的平均时间（清洗时间加上等待时间）和在服务员处所排队列的平均长度。这个系统中的“事件”包括：汽车到达服务员处，汽车被引向汽车清洗机和汽车离开清洗机。图 4-31 概括了这个示例。注意，它包括“源”和“水池”对象，以方便模拟汽车到达和离开工厂。另外注意，当汽车离开汽车清洗机时，必须通知服务员，使得他知道机器是否繁忙。

另外，有时人们期望将一些现存的、串行程序组件（它们以不规则的方式交互）组合为一个并行程序，而不改变组件的内在特征。

针对这样的问题，将每一个组件

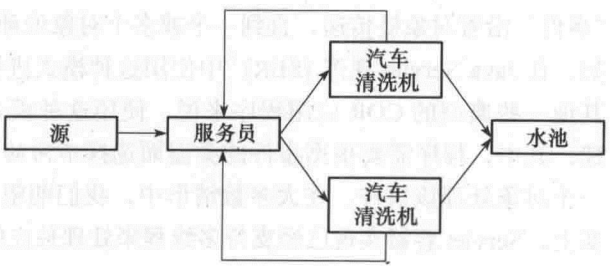



图 4-31 一个汽车清洗工厂的离散事件仿真（箭头表示事件流）

定义为一个任务（或者一个紧密耦合的任务分组）的并行算法是可行的，在离散事件仿真情形中，该组件是一个仿真实体。于是这些任务间的交互基于任务之间的数据流所决定的顺序约束。

3. 面临的问题

- 一个好的解决方案应当能够简单地表示顺序约束，这些顺序约束可能是不规范的，甚至是动态产生的。它还应当使尽可能多的活动能够并发地完成。
- 数据依赖所隐含的顺序约束可以通过将它们编码到程序中来表示（例如，利用串行合成），或者采用共享变量来表示。但是利用两种方法都无法得到简单、能够表示复杂的约束关系和易于理解的解决方案。

4. 解决方案

一个好的解决方案基于使用所谓的事件抽象表示的数据流，其中每一个事件具有一个产生它的任务和一个处理它的任务。因为一个事件必须在处理它之前产生，所以事件也定义任务间的顺序约束。每个任务内的计算由处理这些事件组成。

**定义任务。**每一个任务的基本结构由接收一个事件，处理该事件，以及可能产生的事件组成，如图 4-32 所示。

```
initialize
while(not done)
{
  receive event
  process event
  send events
}
finalize
```

图 4-32 基于事件的协作模式中一个任务的基本结构

如果程序由已有的组件构建，则任务通过向组件提供一个基于事件的一致接口，它将作为正面模式（Facade pattern）[GHJV95] 的一个实例。

115
~
116

任务接收事件的顺序与应用程序的顺序约束必须一致，这一点将在后面讨论。

**表示事件流。**为了实现计算和通信重叠，通常需要某种形式的事件异步通信，在这种通信方式中，一个任务可以创建（发送）一个事件，然后继续向后执行，而不用等待接收者接收它。在消息传递环境中，一个事件可以由从产生该事件的任务异步发送到处理它的任务的一条消息表示。在一个共享内存环境中，可以使用一个队列仿真消息传递。因为每个这样的队列可以被多个任务访问，所以必须以一种能安全并发访问的形式来实现它，如 5.9 节所述。其他的通信抽象，如 Linda 协作语言或 JavaSpace [FHA99] 中的元组空间（tuple space），也可以有效地使用这种模式。Linda [CG91] 是一种仅包含 6 种操作的简单语言，这些操作完成对一个称为元组空间的相关联（即内容可寻址）共享内存的读和写。元组空间是一种概念上的共享存储区，用于存放那些包含称为元组的对象的数据，其中，元组由任务使用，用于在一个分布式系统中通信。

**强制执行事件的顺序。**顺序约束的强制执行在以下情形中是必需的：如某个任务需要以一种与事件的发送顺序不同的顺序来处理这些事件，或者一个任务在处理一个事件时，需要

等待，直到接收到一个给定任务的其他事件，因此通常需要能够预测队列或消息缓冲区，并且能够以任意顺序移除元素。例如，考虑图 4-33 中描述的情形。任务 1 产生了一个事件，将它发送给任务 2，任务 2 将处理该事件，并且任务 1 也将该事件发送给任务 3，其中，任务 3 记录所有事件的信息。任务 2 处理来自于任务 1 的事件，并产生一个新事件，新事件的副本也被发送给任务 3。假设由于调度和底层通信层的不确定性导致来自于任务 2 的事件比来自任务 1 的事件早到达。依赖于任务 3 处理这些事件的方式，这种情形可能会产生问题，也可能不会产生问题。如果任务 3 只是简单地记录所产生的事件的数目，则不会产生问题。但是，如果任务 3 写一个日志记录项，该记录反映事件的处理顺序，则按照事件的到达顺序简单地处理它们，将会产生不正确的结果。如果任务 3 控制一个门，来自于任务 1 的事件造成门打开，来自于任务 2 的事件造成门关闭，因此，顺序凌乱的消息将造成严重的问题，直到来自于任务 1 的事件到达并被处理之后，任务 3 才能处理第一个事件。

在离散事件仿真中，因为应用程序域的语义而出现一些类似的问题。一个事件在当它应当被调度时，在某个仿真时间该事件到达一个站（任务）。一个事件可以先于其他具有较早仿真时间的事件到达一个站。

在某些特定情形中，第一步是确定顺序凌乱的事件是否会产生问题。如果“事件”的路径是线性的，那么顺序凌乱的事件就不会发生，也将不存在问题（或者根据应用的语义，顺序凌乱的事件不会产生问题）。

如果顺序凌乱的事件出现问题，无论是乐观的还是悲观的方式都可以选择。乐观方式需要有对事件错误执行造成的影响执行回滚的能力（包括因为乱序执行所创建的新事件带来的影响）。在分布式仿真领域，这种方法称为时间扭曲（time wrap）[Jef85]。如果一个事件与外部世界产生交互，则乐观方式通常行不通。悲观方式是以增加延迟和通信开销为代价，确保事件总是按照顺序执行。悲观方式需要确保足够“安全”才可以执行事件。例如，在图 4-33 中，直到任务 3 确保没有来自任务 1 的更早事件到达，任务 3 才处理来自于任务 2 的事件，反之亦然。向任务 3 提供该知识可能需要引入一些空事件，除了包含事件的顺序之外，这些空事件不包含任何有用信息。悲观方式的许多实现基于时间戳，时间戳与系统中的因果关系一致 [Lam78]。

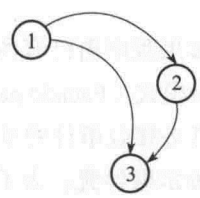



图 4-33 3 个任务间基于事件的通信（以响应从任务 1 接收的事件，任务 2 产生自己的一个事件。这两个事件被发送给任务 3，并可以任意顺序到达任务 3）

大量的研发工作同时采用乐观方式 [RMC+98] 和悲观方式 [CLL+99] 处理离散事件仿真中的事件顺序。类似地，在通信系统所引起的进程组中，可以利用中间件处理事件顺序问题。其中一个示例是由 Cornell [vRBH+98] 所开发的 Ensemble 系统。

**避免死锁**。使用这种模式的系统可能在某个应用层产生死锁，因为某些原因，系统处于一种状态，在该状态中，如果不先从另外一个任务处接收一个永远都不会到达的事件，任务将无法继续执行。这种情形可能由于一个编程错误而引发；在仿真中，它也可能由于被仿真的模型中的问题所产生。对于后一种情形，开发者必须重新考虑解决方案。

如果用悲观技术控制事件的处理顺序，即当一个事件达到，实际上本该被处理，但因为不知道该事件是否安全而没有处理它时，将会发生死锁。通过交换足够的信息来表明事件可

以安全地处理，死锁可以解除。当处理死锁的开销能够抵消并行性的优点，并且使得并行算法慢于一个串行仿真时，这将是一个非常重要的问题。处理这种类型的死锁的方法范围包括从频繁地发送足够的“空消息”来完全避免死锁（代价是需要许多额外的消息），到使用死锁检测策略来检测死锁的存在，然后解决它（代价是在检测和解决死锁之前，可能产生大量的空闲时间）。根据死锁出现的频率来选择相应的方法。一种较好的折衷解决方案是使用超时而不是精确的死锁检测。

117 ~ 118

**调度和处理器分配。**最简单的方法是为每个 PE 分配一个任务，并使得所有的任务并发地执行。如果无法获得充足的 PE，则可以将多个任务分配给一个 PE。这样可确保具有很好的负载平衡。因为潜在不规则的结构和可能的动态属性，以这种模式获得负载平衡是一个非常困难的问题。某些支持这种模式的基础结构允许任务的迁移，使得负载能够在运行时动态地保持平衡。

**事件的高效通信。**如果要使应用程序很好地运行，则用于事件通信的机制必须足够高效。在一个共享内存环境中，这意味着需要确保通信机制不是潜在的瓶颈。在一个消息传递环境中，几种涉及效率的因素需要考虑，例如，在任务间发送很多短消息，或者设法组合这些短消息是否可行。[YWC+96] 和 [WY95] 描述了一些考虑因素与解决方法。

5. 示例

**知名应用。**许多离散事件仿真应用程序使用了这种模式。用于分析空中交通管制系统 [Wie01] 的 DPAT 模拟是一个使用乐观处理技术的成功仿真。它利用 GTW（Georgia Tech Time Warp）系统 [DFP+49] 来实现。论文（[Wie01]）描述了特定于应用程序的调优和几种通用的技术，这些技术使得仿真能够很好地工作，而不用为了乐观的同步花费大量的开销。用于模仿和离散事件仿真的同步并行环境（SPEEDES）[Met] 是另外一种乐观的仿真引擎，它已经用于大规模的作战演习。可扩展仿真框架（SSF）[CLL+99] 是一种具有悲观同步的仿真架构，它已经用于互联网的大规模模拟。

[YWC+96] 中描述的 CSWEB 应用程序模拟了组合数字电路（即无反馈路径的电路）的电压输出。电路首先被分解为多个子电路，与每一个子电路相关的输入信号端口和输出电压端口被连接起来，形成整个电路。每一个子电路的仿真以时间步的模式进行。在每一个时间步，每一个子电路的行为依赖于它前面的状态和从它的输入端口读入的值（这些值对应于其他子电路的输出端口在之前时间步中的对应值）。这些子电路的仿真可以并发地执行，并具有顺序约束，这些约束揭示了输出端口产生的值和输入端口读入的值之间的关系。[YWC+96] 中描述的解决方案符合基于事件的协作模式，它为每一个子电路定义一个任务，并将顺序约束表示为事件。

119

6. 相关模式

这种模式与流水线模式相似，因为两种模式都应用于那些很自然地将计算分解为一个关于半独立实体的集合的问题，并且这些实体间以数据流的形式交互。这两种模式存在两种明显的差别。第一，在流水线模式中，实体之间的交互是相当规范的，流水线的所有阶段以一种松散同步的方式进行，而在基于事件的协作模式中则不存在这种交互，实体以非常不规范和异步的方式交互。第二，在流水线模式中，整体结构（任务数目和任务间的交互）通常是固定的，而在基于事件的协作模式中，问题结构是动态的。

120

第 5 章
Patterns for Parallel Programming

# “支持结构”设计空间

## 5.1 引言

寻找并发性和算法结构设计空间主要集中于算法表达。然而，算法终究要转换为程序。支持结构设计空间描述的编程模式面向并行程序设计阶段，是算法结构设计空间中面向问题模式和实现机制设计空间中特定编程机制间的中间阶段。这些模式描述了支持并行算法表达的软件构造或结构，因此我们将这些模式称为支持结构。图 5-1 描述了支持结构设计空间的总体框图和它在模式语言中的位置。

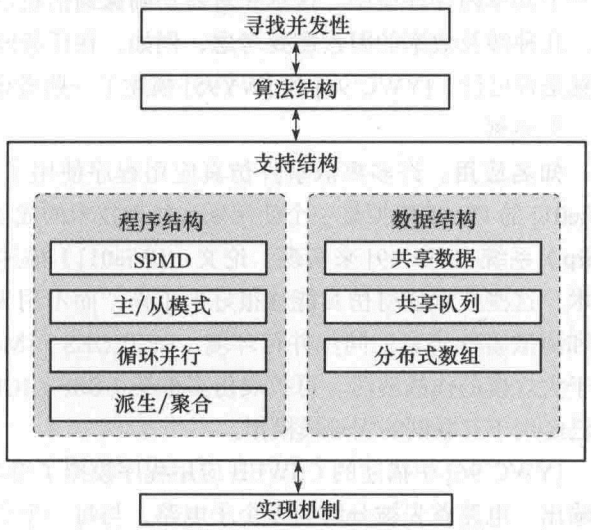


图 5-1 支持结构设计空间的总体框图和它在模式语言中的位置

支持结构设计空间中的模式分为两组：一组表示程序构造方法，另一组表示常用的共享数据结构。下一节简要描述了这些模式。其中一些模式在许多编程环境中得到了很好的支持，这大大减轻了程序员的工作。之所以将它们描述为模式，主要有两个原因：第一，理解这些结构的底层实现细节对于高效使用它们是非常重要的；第二，将这些结构描述为模式为需要重新实现这些模式的程序员提供指南。本章最后一节描述一些结构，或许它们不够重要而没有作为专门模式。但是为了本书描述的完整性，还是有必要介绍它们的。

### 5.1.1 程序结构模式

第一组模式描述了源代码构造方法。这些模式包含如下几种。

- SPMD。在 SPMD（Single Program Multiple Data，单程序多数据）程序中，所有 UE 并行执行同一个程序（单程序），但每个 UE 都拥有自己的私有数据集（多数据）。不同 UE 可按照不同的路径执行程序。在源代码中，可通过使用一个唯一标识 UE（比如，进程 ID）的参数来控制 UE 的执行路径。
- 主 / 从（master/worker）模式。主进程（master）（或主线程）为从进程（worker）（或从线程）建立一个工作池和一个任务包。所有从进程（线程）并发执行，每个从进程（线程）迭代地从任务包中移除一个任务并处理它，直到所有任务都处理完毕或到达某些终止条件为止。在一些实现中，不设立显式的主进程（线程）。

- **循环并行**（loop parallelism）模式。该模式解决了将一个以计算密集循环为主导的串行程序转化为并行程序（循环的不同迭代并行执行）时出现的问题。
- **派生 / 聚合**（fork/join）模式。一个主 UE 派生出多个子 UE，这些子 UE 并行完成全部工作的某一部分。通常派生 UE 处于等待状态，直到所有子 UE 完成任务和执行聚合操作。

121 ~ 122

将这些程序结构定义为不同模式，或多或少存在人为因素。例如，可以使用派生 / 聚合模式或 SPMD 模式实现主 / 从模式。这些模式并没有定义构造并行程序的专有、唯一方式，而是定义经验丰富的并行程序员的主要习惯用法。

根据模式语言所应用的并行编程环境不同，对这些模式的理解也不可避免地存在一些不同。例如，对于 MPI 程序员来说，所有程序结构模式本质上都是 SPMD 模式的变体。而对于 OpenMP 程序员来说，使用线程 ID（SPMD 模式）的程序与使用基于循环的任务共享结构来表达所有并行性的程序（循环并行模式）相比具有很大的差异性。

因此，在使用这些模式时，没有必要严格区分它们。虽然这些模式代表一些值得单独分析的重要技术，但是在实际应用中，可通过不同方式进行组合以满足特定程序的需要。例如，在 SPMD 模式中，我们将讨论如何使用 SPMD 模式来表达基于并行循环的并行算法。这似乎说明了 SPMD 模式与循环并行模式没有本质区别，但反映了 SPMD 模式的灵活性。

### 5.1.2 数据结构模式

第二组模式负责管理数据依赖性。其中，共享数据模式代表一般情况，其他模式描述常用的特定数据结构。

- **共享数据**。该模式解决多个 UE 共享数据时的常见问题，并讨论正确性和性能问题。
- **共享队列**。该模式是熟悉的队列抽象数据类型（ADT）的一种“线程安全”实现，即使被并发执行的多个 UE 共同使用时，该实现也能保证正确的语义。
- **分布式数组**。这种模式代表在并行科学计算中常见的一类数据结构，即一维或者多维数组。这些数组被划分成多个子数组，并在进程或线程间进行分配。

## 5.2 面临的问题

所有程序结构模式解决了相同的基本问题：如何构造源代码以更好的支持所选择的算法结构。唯一面临的问题是每种模式的适用性以及围绕这些结构设计一个程序时需要考虑的常见问题。

- **抽象清晰性**。并行算法能从源代码中清晰地显示出来吗？ 123

一个结构清晰的程序中，算法跃然纸上，读者可非常容易地了解算法的细节，我们称之为抽象清晰。因为并行程序员必须处理多个彼此存在交互的并发任务，所以抽象清晰性对于编写正确的代码（特别是并行程序）非常重要。当通过源代码很难指出算法结构时，做到这一点是非常困难的。

- **可扩展性**。并行程序可高效应用于多少个处理器？

程序的可扩展性受三个因素限制。首先，算法必须要有足够的并发性。如果在超过 10 个 PE 上运行一个仅有 10 个并发任务的算法，将得不到任何好处。其次，算法中串行任务的运

行时间比例将限制所能够使用的处理器数目。第 2 章使用 Amdahl 定理定量地讨论了这一点。最后，算法的并行开销增加了 Amdahl 定理中的串行时间比例，进而限制了算法的可扩展性。

- **效率**。程序对并行计算机资源的有效利用率是多少？我们回忆一下第 2 章给出的效率的量化定义：

$$E(P) = \frac{S(P)}{P} \tag{5-1}$$

$$= \frac{T(1)}{PT(P)} \tag{5-2}$$

式中，$P$ 是 PE 数目，$T(1)$ 是串行参考时间，$T(P)$ 是 $P$ 个 PE 的并行时间。$S(P)$ 为加速比。

效率最严格的定义将 $T(1)$ 设置为在研究的并行算法对应的最佳串行算法的执行时间。然而，当分析并行程序时，并不是每次都能获得“最佳”的串行算法，所以通常使用并行程序在单个 PE 上的运行时间作为参考时间。因为即使当并行程序运行在单个 PE 上时，管理并行计算也总会产生一些开销，所以可能会导致得出的效率比真实效率高。效率与可扩展性密切相关，一个高度可扩展的算法也是高效的算法。然而，当算法的可扩展性被可用的任务数目或并行硬件所限制时，算法效率也可能不受此影响。

- **可维护性**。程序容易调试、验证和修改吗？

将算法转换为源代码几乎从来不是一次就能完成的。例如，程序可能需要调试、添加新功能以及调优性能等，对代码的这些重构称为*代码维护*。代码的可维护性取决于对代码完成
124 这些改变以及进行正确性验证的难易程度。

- **环境适应性**。程序能和所选的编程环境以及硬件很好地结合在一起吗？

例如，如果硬件不支持共享内存，那么基于共享内存的算法结构将不是一个好的选择。编程环境的选择也会产生此类问题。当创建编程环境时，创建者通常会在大脑里构思具有某种特定类型的编程方式。例如，OpenMP 是专门为由大量循环构成的程序设计的，循环迭代将会被多个线程分割以并行执行（并行循环模式）。当程序结构能够和编程环境很好地结合时，软件编写也相对容易。

- **串行等价性**。一个程序在多个 UE 上的运行结果和在一个 UE 上的运行结果相同吗？如果不相同，它们又具有什么关系？

无论并行程序运行在多少个 PE 上，人们都期望每次执行结果都应该是相等的。但事实并不总是这样，特别是需要执行按位相等操作时，由于浮点数运算会以不同的顺序执行可能会导致结果产生小的改变（对于病态算法来说，改变可能会非常大）。如果并行程序在一个处理器上执行多次都能给出相同的结果，则可以推断程序是正确的，并且需要在单个处理器上进行大部分程序测试。如果可能，串行等价性是一个非常理想的目标。

## 5.3 模式选择

程序结构选择通常会比较简单。在大多数情况下，为工程所选的编程环境以及所使用的*算法结构*设计空间模式将会给出合适的程序结构模式。我们将独立考虑这两个因素。

*算法结构*模式与*支持结构*模式的对应关系如表 5-1 所示。注意，*支持结构*模式可应用于多个*算法结构*模式。例如，考虑主 / 从模式：在 [BCM+91.CG91.CGMS94] 中，从极易并行

程序（任务并行模式中的一种特殊情况）到使用几何分解模式的程序，主/从模式都有应用。SPMD 模式更加灵活，覆盖了在科学计算（科学计算一般强调几何分解、任务并行以及分治模式）中使用的大多数重要的算法结构。这种灵活性也使得完全基于算法结构模式选择一种程序结构模式变得非常困难。

表 5-1 支持结构模式与算法结构模式之间的对应关系。星的数目（0 ~ 4）表示对于给定的支持结构模式，实现对应的算法结构模式的可能性

| | 任务并行 | 分治 | 几何分解 | 递归数据 | 流水线 | 基于事件的协作 |
|---|---|---|---|---|---|---|
| SPMD | ★★★★ | ★★★ | ★★★★ | ★★ | ★★★ | ★★ |
| 循环并行 | ★★★★ | ★★ | ★★★ | | | |
| 主/从模式 | ★★★★ | ★★ | ★ | ★ | ★ | ★ |
| 派生/聚合模式 | ★★ | ★★★★ | ★★ | | ★★★★ | ★★★★ |

然而，编程环境的选择有助于缩小选择范围。表 5-2 给出了编程环境和支持结构模式之间的对应关系。例如，对于在分布式内存计算机上使用的 MPI 编程环境来说，SPMD 是无疑是最佳选择。而在共享内存计算机上使用的 OpenMP 编程环境与循环并行模式结合得最好。编程环境和算法结构模式的组合通常能够选择一种支持结构模式。

表 5-2 编程环境和支持结构模式之间的对应关系。星的数目（0 ~ 4）表示对于给定的支持结构模式，可在对应的编程环境中使用的可能性

| | OpenMP | MPI | Java |
|---|---|---|---|
| SPMD | ★★★ | ★★★★ | ★★ |
| 循环并行 | ★★★★ | ★ | ★★★ |
| 主/从模式 | ★★ | ★★★ | ★★★ |
| 派生/聚合模式 | ★★★ | | ★★★★ |

## 5.4 SPMD 模式

1. 问题

编写正确高效的并行程序会遇到很多问题，绝大多数问题是由 UE 间的交互导致的。程序员如何构造并行程序，使得这些交互更具有可管理性，并能更方便地与核心计算结合在一起呢?

2. 背景

并行程序将编写程序的复杂性提高到一个新的级别。编写其他程序所遇到的所有普遍性挑战，编写并行程序都会遇到。编写并行程序的最大挑战是程序员必须同时管理运行在多个 UE 上的多个任务。此外，这些任务和 UE 的交互，要么通过消息传递，要么通过共享内存来实现。尽管并行程序具有这些复杂性，但是一方面必须要保证程序的正确性；另一方面，为了避免过度的额外开销，因此必须要充分协调任务间的交互。

125 ~ 126

幸运的是，大多数并行算法在每一个 UE 上都执行相似的操作。虽然每个 UE 处理的数据可能不一样，或者在某些 UE 上可能会执行一些稍微不同的计算（例如，偏微分方程解法器对边界条件的处理），但是绝大多数情况下，每一个 UE 会执行相似的计算。因此，在很多

场景中，将任务放在同一源树中，可使任务和它们之间的交互更具管理性。采用这种方式，可统一任务的逻辑以及任务间交互的逻辑，因此更容易管理它们。

这就是所谓的“单程序多数据”（SPMD）方法。在可扩展并行计算发展的前期，这是构造并行程序的主要方式。很多编程环境，特别是 MPI，都支持这种方法[⊖]。

除对程序员有优势之外，SPMD 也方便对解决方案的管理。如果仅需要管理一个程序，那么保持软件基础架构的更新和一致性是非常容易的。对于具有大量 PE 的系统来说，这个因素尤其重要。PE 的数量可以非常巨大。例如，根据 2003 年 11 月世界超级计算机 Top 500 列表，世界上最快的两台计算机分别是位于日本地球模拟中心的 Earth Simulator 和位于洛斯阿拉莫斯国家实验室的 ASCI Q，它们分别具有 5120 和 8192 个处理器。如果在每个 PE 上运行一个不同的程序，那么应用软件的管理就会变得非常困难。

SPMD 是构造并行程序的最常用模式。对于 MPI 程序员以及需要使用任务并行模式和几何分解模式求解的问题来说，更是如此。而对于使用分治模式和递归数据模式求解的问题，SPMD 也被证明是非常高效的。

3. 面临的问题

- 对于程序员来说，在每一个 UE 上运行相似的代码，是比较容易的事情。但是大多数复杂的应用程序往往需要在不同的 UE 上运行不同的操作，并且处理不同的数据。
- 软件的生命周期往往比给定的并行计算机长。因此，程序应当具有可移植性。这就强迫程序员假设编程环境中的最低通用标准，并假设仅可以利用基本机制来协作任务
127 交互。
- 并行程序要获得高可扩展性以及卓越性能需要程序与并行计算机架构很好地结合在一起。因此，程序员必须掌握以及合理控制并行计算系统的架构细节。

4. 解决方案

SPMD 模式通过创建运行在每个 UE 上的源代码副本，很好地解决了这些问题。解决方案由下面的基本元素构成。

- **初始化**。程序被加载到每一个 UE 上，并打开 bookkeeping 操作以创建一个通用上下文。这个过程的细节与并行编程环境紧密集合，并且通常会建立其他 UE 的通信通道。
- **获得唯一标识符**。在程序的最上面，设置一个唯一标识 UE 的标识符。它通常是 MPI 组范围内 UE 的秩（即 0 ~ $N$-1 之间的一个数字，其中 $N$ 是 UE 的数目），或者 OpenMP 中的线程 ID。这个唯一标识符可使不同 UE 在程序运行期间制定不同的策略。
- **在每个 UE 上运行相同的程序，并使用唯一的 ID 来区分不同 UE 的行为**。每个 UE 上运行相同的程序，其所执行指令的差别通常由标识符来定义（也可以依赖于 UE 的数据）。不同 UE 执行源代码中不同路径的选择方式有多种，最常用的有：①使用分支语

⊖ 这并不是编程环境推出 SPMD 的原因，主要原因来源于其他方面。MIMD（Multiple Instructions Multiple Data，多指令多数据）机器的编程环境推出 SPMD 是因为 SPMD 是程序员编写程序的最好方式。之所以采用这种编程方式，是因为 SPMD 既可以获得正确的逻辑、高效的任务执行和交互，也支持在不同的 UE 上运行不同的程序（有时候这称为 MPMD 程序结构）。根据 PVM 的经验和优点，MPI 设计者选择仅支持 SPMD。

句，将特定的代码块赋予不同的 UE；②使用 UE 的标识符来计算循环索引，从而使不同的 UE 执行不同的迭代。

- **分配数据**。UE 上的数据分配主要有两种方式：①全局数据分块，并存储在对应 UE 的局部内存中，如果需要，再将它们组合为全局相关结果。②共享或者复制程序的主要数据结构，并使用 UE 标识符将数据子集与对应的 UE 相关联。
- **结束**。通过清除共享上下文并终止计算来结束程序。如果全局相关数据分布在各个 UE 中，则需要重新组合。

**讨论**。开发 SPMD 程序时，一个需要牢记的重要问题是*抽象的清晰性*，即通过阅读程序的源代码来理解算法的难易程度。这依赖于数据的处理方式，处理方式的好坏将直接影响结果。如果需要使用 UE 标识符进行复杂的代数运算，以确定与 UE 相关的数据或指令分支，则几乎无法从源代码中识别出算法（5.10 节讨论关于数组的有用技术）。 128

在很多情况下，结合简单循环分割方法的数据复制算法是最佳选择。这是因为这种方法不仅实现了并行算法源代码抽象的清晰性，而且实现了与串行算法的高度等价性。遗憾的是，这种简单方法的可扩展性比较差，因此需要更加复杂的解决方案。实际上，SPMD 算法具有高可扩展性并且需要 UE 间的复杂协作。SPMD 模式已经编写了能够扩展到几千个 UE[PH95] 上的算法。高度可扩展算法通常极度复杂，因为它们需要在节点间分配数据（也就是说，不存在简单的数据复制技术），并且通常会使用复杂的负载均衡逻辑。遗憾的是，这些算法在它们的连续副本之间缺乏相似性，这也反映了 SPMD 模式的一个常见缺点。

SPMD 的一个重要优势就是与启动和终止相关的开销都隔离在程序的开始和结束之处，而不位于以时间为关键因素的循环之内。这提高了程序效率，同时也引发了由通信开销、UE 间计算负载均衡以及算法可用并发量驱动的效率问题。

SPMD 程序能与基于消息传递的编程环境紧密结合。例如，大多数 MPI 或 PVM 程序使用 SPMD 模式。但是，要注意，OpenMP 也可能使用 SPMD 模式 [CPP01]。在硬件方面，SPMD 不假设关于地址空间（任务在该地址空间范围内执行）的任何内容。只要每个 UE 能够运行指令流来操作其数据（即，计算机可以归类为 MIMD 类型），SPMD 结构就满足要求。SPMD 程序的这种通用性是一种强大优势。

5. 示例

采用 SPMD 模式的应用程序所引发的问题可用如下 3 个特殊的示例来讨论：

- 数值积分，使用梯形法则来估算一个定积分的值；
- 分子动力学、作用力计算；
- Mandelbrot 集合计算。

**数值积分**。本节使用在并行程序教学中频繁用到的一个非常简单的程序来探讨 SPMD 模式所带来的问题。考虑使用式（5-3）估计 $\pi$ 的值：

$$\pi = \int_0^1 \frac{4}{1+x^2} \mathrm{d}x \qquad (5\text{-}3)$$

我们使用梯形法则来求解这个积分。其基本思想是使用一系列矩形填充一条曲线之下的区域。当矩形的宽度接近 0 时，矩形面积的和接近于积分值。 129

图 5-2 显示了在单处理器上进行这种计算的串行程序。为简单起见，我们将积分中的迭

代次数固定为1000 000。变量sum初始化为0，步长由x（在这种情形中，x为1.0）除以总迭代次数计算得到。每一个矩形的面积等于宽（步长）乘以高（被积函数在区间的中心值）。因为宽是一个常量，所以我们将其放在迭代循环之外，并使用步长（step）乘以矩形高度的总和，以获得定积分的估算值。

```
#include <stdio.h>
#include <math.h>

int main () {
   int i;
   int num_steps = 1000000;
   double x, pi, step, sum = 0.0;

   step = 1.0/(double) num_steps;

   for (i=0;i< num_steps; i++)
   {
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
   printf("pi %lf\n",pi);
   return 0;
}
```

图5-2 利用梯形规则估算π值的串行程序

我们将会看到这个算法的不同并行版本，图5-3所示。从这个算法的简单MPI版本中，可以看到经典SPMD程序的所有元素。相同的程序运行在每个UE上。在程序的初始部分，初始化MPI环境，每个UE的ID（my_id）由与communicator MPI_COMM_WORLD（关于communicator和其他MPI的细节信息，参考附录B）进程组中的每个UE的进程秩所给定。根据UE的数量和ID，将某个范围的（i_start和i_end）循环分配给UE执行。因为循环迭代次数可能不能被UE数量整除，所以必须保证最后一个UE运行完循环中的最后一次迭代。每个UE计算出部分和之后，乘以步长step，然后使用MPI_Reduce()例程接口将部分和归约为一个全局结果（第6章非常详细地描述归约操作）；这个全局结果只能在my_id==0的进程中获得，因此，我们用此进程输出最终结果。

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
   int i, i_start, i_end;
   int num_steps = 1000000;
   double x, pi, step, sum = 0.0;

   int my_id, numprocs;
   step = 1.0/(double) num_steps;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
```

图5-3 利用梯形规则积分的MPI并行程序，其方式是将循环迭代分块分配给每个UE，最后执行归约操作

```
    i_start = my_id * (num_steps/numprocs);
    i_end = i_start + (num_steps/numprocs);
    if (my_id == (numprocs-1)) i_end = num_steps;

    for (i=i_start; i< i_end; i++)
    {
       x = (i+0.5)*step;
       sum = sum + 4.0/(1.0+x*x);
    }
    sum *= step;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
       MPI_COMM_WORLD);
    if (my_id == 0) printf("pi %lf\n",pi);
    MPI_Finalize();
    return 0;
}
```

图 5-3 （续）

实质上，图 5-3 中的示例中所做的是复制关键数据（在这里是部分和，sum），根据 UE 的 ID 将任务显式地划分为快，每个 UE 处理一个任务块，然后将局部结果归约为最终结果。这种模式具有三个挑战：①正确划分数据，②正确归约结果，③均匀分配工作。前两个挑战在这个示例中微不足道。然而，负载平衡稍微有些困难。在图 5-3 的示例中，如果循环迭代次数不能整除 UE 数目，可能会导致最后一个 UE 分配较多的工作。为了更均衡地分配工作，需要将额外的循环迭代分配给多个 UE。图 5-4 中的程序片段给出了一种完成这种功能的方法：将循环迭代次数除以处理器数目后，计算剩余的迭代次数（rem），然后由前 rem 个 UE 负责完成这些循环迭代。然而，这种索引计算方式也是 SPMD 模式程序员的潜在危险点，不仅非常容易出错，而且当程序员试图理解这种逻辑推理时，非常耗时间。

```
int rem = num_steps % numprocs;

i_start = my_id * (num_steps/numprocs);
i_end = i_start + (num_steps/numprocs);

if (rem != 0){
   if(my_id < rem){
      i_start += my_id;
      i_end += (my_id + 1);
   }
   else {
      i_start += rem;
      i_end += rem;
   }
}
```

图 5-4 当迭代次数不能被 UE 数目整除时，这种索引计算方式能够均匀分配任务。其思想是将剩余的任务（rem 个）分配给前 rem 个 UE

最后，我们在数值积分程序中应用一种循环划分策略，最终程序如图 5-5 所示。这是一种常用的任务划分技巧，即周期性地分配循环迭代：每个 UE 从等于它的秩的迭代开始，以等于 UE 数目的跨度在循环迭代中前进。循环迭代像分发纸牌所采用的方法一样被交错分配给 UE。这个程序版本均衡地分配了任务负载，而不用求助于复杂的索引计算。

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
   int i;
   int num_steps = 1000000;
   double x, pi, step, sum = 0.0;

   int my_id, numprocs;
   step = 1.0/(double) num_steps;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

   for (i=my_id; i< num_steps; i+= numprocs)
   {
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
   sum *= step;
   MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
   if (my_id == 0) printf("pi %lf\n",pi);
   MPI_Finalize();
   return 0;
}
```

图 5-5 利用梯形规则积分的 MPI 并行程序，使用一种简单的循环划分算法：周期性分配迭代，最后执行一个归约操作

SPMD 程序也可以用 OpenMP 和 Java 编写。图 5-6 是上述算法的 OpenMP 版本，与 MPI 程序非常相似。该程序具有单个并行区域，我们从确定线程数目以及线程 ID 开始，使用相同的方式将循环迭代交错分布到线程组中。像 MPI 程序一样，使用归约操作将部分和归约为一个全局结果。

```
#include <stdio.h>
#include <math.h>
#include <omp.h>

int main () {
   int num_steps = 1000000;
   double pi, step, sum = 0.0;

   step = 1.0/(double) num_steps;

#pragma omp parallel reduction(+:sum)
   {
      int i, id = omp_get_thread_num();
      int numthreads = omp_get_num_threads();
      double x;

      for (i=id;i< num_steps; i+=numthreads){
         x = (i+0.5)*step;
         sum += + 4.0/(1.0+x*x);
      }
   } // end of parallel region
   pi = step * sum;
   printf("\n pi is %lf\n",pi);
   return 0;
}
```

图 5-6 利用梯形规则积分的 OpenMP 并行程序，使用了同图 5-5 一样的 SPMD 算法

**分子动力学**。本节使用分子动力学算法作为一个递归示例介绍这种模式语言。分子动力学仿真一个大型分子系统的运动。该算法使用显式时间步方法，即计算出每个原子在每个时间步的作用力，然后使用经典力学中的标准算法来计算作用力是如何改变原子运动的。

根据目标计算机系统和程序的使用意图，这个算法有多种实现方式，因此该算法对说明并行算法中的关键概念是非常理想的。在本章的讨论中，我们将采用 [Mat95] 中的方法，并假设：①程序已有串行版本；②有一个既可以串行执行也可以并行执行的程序，这点很重要；③目标计算机系统是一个由标准以太网 LAN 连接的小型集群。[PH95] 中讨论了一个可在大规模并行计算机系统中运行并具有高可扩展性的算法。

该算法的核心算法和伪代码，见 3.1.3 节，这里将不再重复讨论。图 5-7 提供了该算法伪代码的一个副本。

```
Int const N // number of atoms

Array of Real :: atoms (3,N) //3D coordinates
Array of Real :: velocities (3,N) //velocity vector
Array of Real :: forces (3,N) //force in each dimension
Array of List :: neighbors(N) //atoms in cutoff volume

loop over time steps
   initialize_forces (N, Forces)
   if(time to update neighbor list)
       neighbor_list (N, Atoms, neighbors)
   end if
   vibrational_forces (N, atoms, forces)
   rotational_forces (N, atoms, forces)
   non_bonded_forces (N, atoms, neighbors, forces)
   update_atom_positions_and_velocities(
                  N, atoms, velocities, forces)
   physical_properties ( ... Lots of stuff ... )
end loop
```

图 5-7 分子动力学示例伪代码。这个伪代码和之前讨论的版本非常相似，但包含了少数特别的细节。为了支持更详细的伪代码示例，初始化作用力数组的函数调用被显式化。同时，邻居列表仅被偶尔更新的事实也被显式化了

该并行算法在第 3 章和第 4 章的几种模式中都讨论过，要点如下。

- `non_bonded_force` 计算是主要的时间开销。
- 计算 `non_bonded_force` 时，每个原子与其他原子都存在潜在交互。因此，每个 UE 需要访问全局原子位置数组。此外，根据牛顿第三定律，每个 UE 将分别求取对整个作用力数组的贡献（见 3.6 节的示例）。
- 集中于一个特定原子所需要的计算是分解 MD 问题任务的一种方式。也就是说，我们可以通过将原子分配给 UE 来并行化这个问题（见 3.2 节的示例）。

假设算法的目标计算机系统是一个小型集群，根据第一点，我们仅并行化作用力的计算。对于并行计算来说，小型集群的网速较慢，并且根据第 2）点中提到的数据依赖性，我们将：

- 在每个节点上存储全局作用力数组和坐标数组的一个副本。
- 每个 UE 冗余地更新每个原子的新位置和速度（相对于并行计算这些数据并进行通信，冗余计算要廉价得多）。

- 每个 UE 计算它对作用力数组的贡献，然后将 UE 的贡献归约进在每个 UE 上复制的单个全局作用力数组。

这个算法是对串行算法的一个简单转换。SPMD 程序的伪代码如图 5-8 所示。与任何 MPI 程序一样，MPI 头文件位于程序的顶部。初始化 MPI 环境，并且 ID 与 MPI 进程的秩相关联。

```
#include <mpi.h>

    Int const N // number of atoms
    Int const LN // maximum number of atoms assigned to a UE

    Int ID // an ID for each UE
    Int num_UEs // the number of UEs in the parallel computation

    Array of Real :: atoms (3,N) //3D coordinates
    Array of Real :: velocities (3,N) //velocity vector
    Array of Real :: forces (3,N) //force in each dimension
    Array of Real :: final_forces(3,N) //globally summed force
    Array of List :: neighbors(LN) //atoms in cutoff volume
    Array of Int :: local_atoms(LN) //atoms for this UE

    ID = 0 // default ID (used by the serial code)
    num_UEs = 1 // default num_UEs (used by the serial code)
    MPI_Init()
    MPI_Comm_size(MPI_COMM_WORLD, &ID)
    MPI_Comm_rank(MPI_COMM_WORLD, &num_UEs)

    loop over time steps
        initialize_forces (N, forces, final_forces)
        if(time to update neighbor list)
           neighbor_list (N, LN, atoms, neighbors)
        end if
        vibrational_forces (N, LN, local_atoms, atoms, forces)
        rotational_forces (N, LN, local_atoms, atoms, forces)
        non_bonded_forces (N, LN, atoms, local_atoms, neighbors,
           forces)

        MPI_All_reduce{forces, final_forces, 3*N, MPI_REAL,
           MPI_SUM, MPI_COMM_WORLD)

        update_atom_positions_and_velocities(
                        N, atoms, velocities, final_forces)
        physical_properties ( ... Lots of stuff ... )
    end loop

    MPI_Finalize()
```

图 5-8 SPMD 分子动力学 MPI 程序的伪代码

该算法仅对串行函数作了少量改动。第一，定义一个用于存储全局作用力的数组 final_forces，以更新原子位置和速度。第二，创建一个列表，包含所有分配给 UE 的原子，该列表将传递给任何将要并行执行的函数。最后，修改 neighbor_list，仅存储分配给 UE 的原子。

最后，在每一个将要并行执行（作用力计算）的函数的内部，作用于 atoms 上的循环被作用于本地原子列表的循环所代替。

图 5-9 给出了关于这些简单改变的代码示例。这个示例几乎与在 4.4 节中讨论的串行版本一致。相对于前面的讨论，主要改变如下。

```
function non_bonded_forces (N, LN, atoms, local_atoms,
                                    neighbors, Forces)

    Int N // number of atoms
    Int LN // maximum number of atoms assigned to a UE

    Array of Real :: atoms (3,N) //3D coordinates
    Array of Real :: forces (3,N) //force in each dimension
    Array of List :: neighbors(LN) //atoms in cutoff volume
    Array of Int :: local_atoms(LN) //atoms assigned to this UE
    real :: forceX, forceY, forceZ

    loop [i] over local_atoms

        loop [j] over neighbors(i)
            forceX = non_bond_force(atoms(1,i), atoms(1,j))
            forceY = non_bond_force(atoms(2,i), atoms(2,j))
            forceZ = non_bond_force(atoms(3,i), atoms(3,j))
            force(1,i) += forceX; force(1,j) -= forceX;
            force(2,i) += forceY; force(2,j) -= forceY;
            force(3,i) += forceZ; force(3,j) -= forceZ;
        end loop [j]

    end loop [i]
end function non_bonded_forces
```

图 5-9 典型并行分子动力学代码中 nonbounded 计算的伪代码。这段代码几乎与图 4-4 所示函数的串行版本一致。仅有的改变是定义了一个新的整型数组 local_atoms，用于存储分配给该 UE 的原子索引。我们也假设创建的邻居列表也仅存储分配给该 UE 的原子。为了给这些数组分配空间，需要添加一个参数 LN，表示可能分配给单个 UE 的最大原子数目

- 增加了一个新数组，用于存储分配给该 UE 的原子的索引。这个数组的长度为 LN，LN 为可能分配给单个 UE 的最大原子数目。
- 作用于所有原子的循环被作用于 local_atoms 列表元素的循环所代替。
- 假设邻居列表已被修改，对应于 local_atoms 列表中的原子。

通过将 LN 设置为 N，并将整个原子索引集合放置到 local_atoms 中，该代码可修改为这个程序的一个串行版本。这个特征符合其中一个设计目标：存在一份源代码，既可以串行执行，也可以并行执行。

这个算法的关键是函数中邻居列表的计算。邻居列表函数包含一个作用于原子的循环。对于每个原子 i，有一个作用于其他所有原子的循环和测试，以确定哪些原子是原子 i 的邻居。这些邻居原子的索引保存在 neighbors（一个关于列表的列表）中。伪代码如图 5-10 所示。

```
function neighbor (N, LN, ID, cutoff, atoms, local_atoms,
                   neighbors)

    Int N // number of atoms
    Int LN // max number of atoms assigned to a UE
    Real cutoff // radius of sphere defining neighborhood
```

图 5-10 邻居列表计算的伪代码。对于每个原子 i，半径为 cutoff 的球形范围内的所有原子的索引都被添加到原子 i 的邻居列表中。注意，第二层循环仅考虑了索引大于 i 的原子，这是因为根据牛顿第三定律，作用力是相互的，原子 i 对原子 j 的作用力是原子 j 对原子 i 作用力的负数

```
    Array of Real :: atoms (3,N) //3D coordinates
    Array of List :: neighbors(LN) //atoms in cutoff volume
    Array of Int :: local_atoms(LN) //atoms assigned to this UE
    real :: dist_squ

    initialize_lists (local_atoms, neighbors)

    loop [i] over atoms on UE //split loop iterations among UEs
        add_to_list (i, local_atoms)
        loop [j] over atoms greater than i
            dist_squ = square(atom(1,i)-atom(1,j)) +
                       square(atom(2,i)-atom(2,j)) +
                       square(atom(3,i)-atom(3,j))
            if(dist_squ < (cutoff * cutoff))
                add_to_list (j, neighbors(i))
            end if
        end loop [j]

    end loop [i]
end function neighbors
```

图 5-10（续）

UE 间进行分配的并行逻辑位于图 5-10 的单个循环中。

```
loop [i] over atoms on UE //split loop iterations among UEs
    add_to_list (i, local_atoms)
```

该循环在 UE 间的划分方式依赖于具体的编程环境。对于 MPI 来说，一种非常好的方式是图 5-5 讨论的周期分配。

```
for (i=id;i<number_of_atoms; i+= number_of_UEs){
    add_to_list (i, local_atoms)
}
```

通过创建一个 owner-computes 过滤器 [Mat95]，就可以引入一个更为复杂的分配方式，甚至可以是动态分配方式。owner-computes 过滤器提供了一种灵活且可重用的调度机制将循环迭代映射到 UE 上。该过滤器是关于 ID 和循环迭代的一个布尔函数。函数的值依赖于 UE 是否“拥有”循环的一个特定迭代。例如，在分子动力学程序中，把 owner-computes 函数的调用添加到一个作用于原子的并行循环的顶部。

```
for (i=0;i<number_of_atoms; i++){
    if !(is_owner (i)) break

    add_to_list (i, local_atoms)
}
```

为支持并行性的表达，不需要再对循环做其他改变。如果管理循环的逻辑是复杂的，这种将循环迭代分配给 UE 的方法不需要改变这个逻辑，并且索引划分逻辑非常清晰地位地于源代码的某个位置。当几个应当采用相同方式调度的循环在程序中展开时，会产生另一个优点。例如，在一台 NUMA 机器或者一个集群上，分配给一个 PE 的数据尽可能地多次使用是非常重要的。通常，这意味着在多个循环中重用相同的调度机制。

这种方法在 [Mat95] 中的分子动力学应用程序中有更加详细的描述。在这个应用程序中，这是非常重要的，这是因为邻居列表生成所产生的工作负载可能无法精确反映各种作用力的计算负载。如果能非常容易地收集关于每个原子所需要时间的信息，并重新调整 `is_owner` 函数，就可以产生更优的负载调度。

这些 SPMD 算法同样适用于 OpenMP 程序。所有的基本函数保持不变，改变最顶层的程序以迎合 OpenMP 程序的需求，如图 5-11 所示。

```
#include <omp.h>
    Int const N // number of atoms
    Int const LN // maximum number of atoms assigned to a UE
    Int ID // an ID for each UE
    Int num_UEs // number of UEs in the parallel computation
    Array of Real :: atoms(3,N) //3D coordinates
    Array of Real :: velocities(3,N) //velocity vector
    Array of Real :: forces(3,N) //force in each dim
    Array of List :: neighbors(LN) //atoms in cutoff volume
    Array of Int :: local_atoms(LN) //atoms for this UE

    ID = 0
    num_UEs = 1

#pragma omp parallel private (ID, num_UEs, local_atoms, forces) {
    ID = omp_get_thread_num()
    num_UEs = omp_get_num_threads()

    loop over time steps
        initialize_forces (N, forces, final_forces)
        if(time to update neighbor list)
            neighbor_list (N, LN, atoms, neighbors)
        end if
        vibrational_forces (N, LN, local_atoms, atoms, forces)
        rotational_forces (N, LN, local_atoms, atoms, forces)
        non_bonded_forces (N, LN, atoms, local_atoms,
                                        neighbors, forces)
#pragma critical
        final_forces += forces
#barrier

#pragma single
{
        update_atom_positions_and_velocities(
                        N, atoms, velocities, forces)
        physical_properties ( ... Lots of stuff ... )
} // remember, the end of a single implies a barrier

    end loop

} // end of OpenMP parallel region
```

图 5-11 分子动力学 OpenMP 并行程序的伪代码

作用于时间的循环被放置在一个并行区域内。这个并行区域由 parallel prama 语句创建。

```
#pragma omp parallel private (ID, num_UEs, local_atoms, forces)
```

该 pragma 语句创建了一个线程组，线程组的每个成员都执行这个作用于时间的循环。private 语句将对应变量复制到每一个 UE 中。归约操作在一个临界区域进行：

```
#pragma critical
        final_forces += forces
```

不能在并行区域使用 reduction 子句，因为直到并行区域的操作全部完成后，才能得到这个结果。正确结果在临界区内生成，但是算法的运行时间同 UE 的数目呈线性关系。因此，相对于第 6 章讨论的其他归约算法，这种方法不是最优的。但是在具有适量数目处理器的计算机系统中，临界区的归约操作可以很好地工作。

在临界区之后需要一个 barrier 操作，以确保归约操作在原子位置和速度信息更新之前完成。然后使用 OpenMP 的 `single` 结构引发一个 UE 进行更新工作。在 `single` 语句之后不需要 barrier 操作，因为 `single` 构造本身隐含着一个 barrier 操作。用于计算作用力的函数在 OpenMP 和 MPI 程序版本中未发生变化。

**Mandelbrot 集合计算**。本节将讨论知名的 Mandelbrot 集合 [Dou86]。该算法作为一个任务并行问题，在 4.4 节中，已经对该算法及其并行化进行了讨论。每个像素基于式（5-4）的二次递归关系进行着色。

$$Z_{n+1} = Z_n^2 + C \tag{5-4}$$

式中，$C$ 和 $Z$ 是复数，递归从 $Z_0$=$C$ 开始。在垂直轴（–1.5 ~ 1.5）绘制 $C$ 的虚部，在水平轴（–1 ~ 2）绘制 $C$ 的实部。如果递归关系收敛于一个稳定值，则每个像素的颜色都是黑色的；否则，根据递归关系的发散速度对像素进行着色。

在 4.4 节描述的并行算法中，任务粒度为图像中每一行的计算。当任务数目多于 UE 数目时，静态调度策略应该可以在节点间实现高效的静态负载均衡。下面将介绍在 MPI 中如何使用 SPMD 模式来求解这个问题。

图 5-12 为该算法的串行版本伪代码。该算法令人感兴趣的部分隐藏在 `comput_Row()` 例程中。因为这个函数的具体实现细节对于理解并行计算来说并不重要，所以这里将不显示它们。对于一行中的每个点，需要进行如下处理。

```
Int const Nrows // number of rows in the image
Int const RowSize // number of pixels in a row
Int const M // number of colors in color map

Real :: conv // divergence rate for a pixel
Array of Int :: color_map (M) // pixel color based on conv rate
Array of Int :: row (RowSize) // Pixels to draw
Array of Real :: ranges(2) // ranges in X and Y dimensions

manage_user_input(ranges, color_map) // input ranges, color map
initialize_graphics(RowSize, Nrows, M, ranges, color_map)

for (int i = 0; i<Nrows; i++){

    compute_Row (RowSize, ranges, row)

    graph(i, RowSize, M, color_map, ranges, row)

} // end loop [i] over rows
```

图 5-12 Mandelbrot 集合生成程序的串行版本伪代码

- 每个像素对应于二次递归关系式中 $C$ 的一个值，我们将基于输入 range 和像素的索引来计算。
- 计算递归项，并根据它是否收敛于一个固定值来设置像素的值。如果它发散，则根据发散率来设置像素值。

计算完毕后，绘制出所有行，从而绘制出著名的 Mandelbrot 集合。标记像素的颜色根据发散率到颜色表的映射来确定。

如图 5-13 所示，该算法的 SPMD 程序非常简单。我们假设目标计算机系统是某种类型的分布式存储器计算机（一个集群或者一个 MPP），并且有一台机器作为专门的节点用于图形

交互，其他节点用于计算。假设图形节点的秩为 0。

```
#include <mpi.h>
    Int const Nrows // number of rows in the image
    Int const RowSize // number of pixels in a row
    Int const M // number of colors in color map
    Real :: conv // divergence rate for a pixel
    Array of Int :: color_map (M) // pixel color based on conv rate
    Array of Int :: row (RowSize) // Pixels to draw
    Array of Real :: ranges(2) // ranges in X and Y dimensions
    Int :: inRowSize // size of received row
    Int :: ID // ID of each UE (process)
    Int :: num_UEs // number of UEs (processes)
    Int :: nworkers // number of UEs computing rows
    MPI_Status :: stat // MPI status parameter

    MPI_Init()
    MPI_Comm_size(MPI_COMM_WORLD, &ID)
    MPI_Comm_rank(MPI_COMM_WORLD, &num_UEs)

// Algorithm requires at least two UEs since we are
// going to dedicate one to graphics
    if (num_UEs < 2) MPI_Abort(MPI_COMM_WORLD, 1)

    if (ID == 0 ){
       manage_user_input(ranges, color_map) // input ranges, color map
       initialize_graphics(RowSize, Nrows, M, ranges, color_map)
  }

 // Broadcast data from rank 0 process to all other processes
    MPI_Bcast (ranges, 2, MPI_REAL, 0, MPI_COMM_WORLD);
    if (ID == 0) { // UE with rank 0 does graphics
      for (int i = 0; i<Nrows; i++){
        MPI_Recv(row, &inRowSize, MPI_REAL, MPI_ANY_SOURCE,
                      MPI_ANY_TAG, MPI_COMM_WORLD, &stat)
        row_index = stat(MPI_TAG)
        graph(row_index, RowSize, M, color_map, ranges, Row)
      } // end loop over i
    else { // The other UEs compute the rows
      nworkers = num_UEs - 1
      for (int i = ID-1; i<Nrows; i+=nworkers){
        compute_Row (RowSize, ranges, row)
        MPI_Send (row, RowSize, MPI_REAL, 0, i, MPI_COMM_WORLD);
      } // end loop over i
    }
    MPI_Finalize()
```

图 5-13 图 5-12 中 Mandelbrot 集合生成程序的并行 MPI 版本伪代码

如附录 B 所述，MPI 程序从通常的 MPI 初始化开始。秩为 0 的 UE 获取用户输入，然后将其广播到其他 UE 中。接着程序循环处理图像行，当计算完成后，接收计算结果并绘制成图像。该程序使用周期分配算法将循环迭代分配给所有秩不等于 0 的其他 UE，当计算完一行后，将结果发送给图形 UE。

**知名应用**。大量 MPI 程序应用这种模式。对 SPMD 程序和示例的教学式讨论可以在多种 MPI 教科书中找到，如 [GLS99] 和 [Pac96]。应用这种模式的代表性应用包括量子化学 [WSG95]、有限元方法 [ABKP03、KLK+03] 以及 3D 空气动力学 [MHC+99]。

6. 相关模式

SPMD 模式非常常用，可用于实现其他模式。这种模式的许多示例与循环并行模式紧密相关。几何分解模式的大多数应用也使用 SPMD 模式。分布式数组模式实际上是使用 SPMD

130 ~ 142

模式为程序分配数据的一种特例。

## 5.5 主/从模式

1. 问题

当需要在 UE 间动态维持任务负载均衡时，该如何组织程序？

2. 背景

算法的并行效率受并行开销、串行部分和负载均衡影响。一个优秀的并行算法必须处理好这些方面。但有时负载平衡难以实现，甚至负载均衡设计会占算法设计的主导地位。这类问题通常具有下面一种或多种特征。

- 与任务相关的工作负载是高度变化并且不可预测的。如果工作负载可预测，则它们可以分割为开销相等的多个部分，然后静态分配给 UE，并使用 SPMD 模式或者*循环并行*模式进行并行化。但如果它们是不可预测的，则静态分配不会产生最优的负载平衡。
- 程序中计算密集部分的程序结构不能映射到简单循环。如果算法是基于循环的，则通过周期性分配循环迭代或者使用一种动态调度策略，通常可以获得统计学意义上接近于最优的静态负载均衡（例如，在 OpenMP 中，使用 `schedule(dynamic)` 从句）。但若程序中的控制结构比简单循环复杂，则需要更通用的负载均衡方法。
- 并行计算可用 PE 的计算能力在整个计算期间不同，随计算的进展而变化，或者是不可预测的。

某些情况下，如果任务紧耦合（它们需要通信或者读写共享数据）并且必须在同一时间内活跃，主/从模式是不可行的：程序员别无选择，只能按照大小或者分组将任务动态地（即在计算期间）分配给 UE，以获得一种高效的负载平衡。然而，这种方式实现起来可能非常困难，如果不仔细，很可能会增加大量的并行开销。

如果任务相互独立，或者任务依赖性可以通过某种方法排除在并行计算之外，那么程序员在平衡负载时，就具有极大的灵活性。可能会使负载平衡自动完成，这也是这种模式要解决的问题。

*当任务间不存在依赖性时（易并行问题），这种模式特别适用于任务并行模式问题。对于*
143 *任务可以间接映射到 UE 上的情形，也可以使用派生/聚合模式。*

3. 面临的问题

- 每一个任务的工作量，以及在某些情形下甚至 PE 的计算能力都是不可预知的。在这种情况下，显式预测任何给定任务的运行时间是不可能的。因此，平衡均衡设计必须独立于给定的任务。
- 负载均衡操作可能会带来非常昂贵的通信开销。这就意味着任务调度应当围绕着较小数目的大任务进行。但是，大任务减少了可在 PE 间进行分配的任务数目，这又增加获得良好负载平衡的难度。
- 最佳负载均衡算法可能非常复杂而且需要变更程序，这个过程也非常容易出错。程序员需要在最佳负载分配方式和代码易维护性之间进行平衡。

4. 解决方案

面对这些问题，著名的主/从模式是一种很好的解决方案。图 5-14 对这种模式进行了简要说明。这种解决方案由两个逻辑元素组成：一个 master，一个或者多个 worker 实例。

master 初始化计算并设定问题，并创建任务包。在经典算法中，在 worker 任务完成之前，master 都将处于等待状态；全部 worker 任务完成后，master 将收集结果并终止计算。

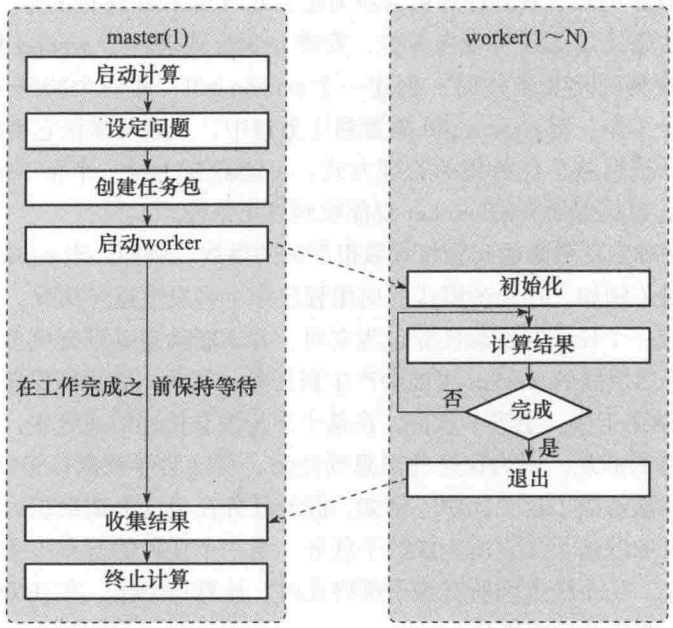



图 5-14 主 / 从模式的两个基本元素是 master 和 worker。master 只有一个，但 worker 可以有若干个。从逻辑上讲，master 设定计算，管理任务包。各个 worker 从任务包中获取任务，执行完成后，返同任务包获取更多任务。这个过程一直持续到终止条件满足为止

master 创建任务包的一种简单方法是利用 5.9 节描述的单个共享队列。当然，也存在许多其他机制来创建一个全局共享结构，可以在其中插入或者移除任务。具体实例包括：元组空间 [CG91、FHA99]、分布式队列或单调计数器（当任务可以用一组连续的整数来指定时）。

同时，每个 worker 都会进入一个循环。在循环顶层，worker 从任务包中取出一个任务，然后处理任务，并检测是否还有其他任务：如果有，则获取下一个任务。直到满足终止条件，这个过程将持续下去。终止条件满足后，master 将被唤醒，收集结果，并终止计算。

主 / 从算法可自动完成负载均衡。因此，程序员无法显式决定哪一个任务将分配给哪一个 UE。这个决定将由 master 动态制定，这是因为在 worker 完成一个任务后，将访问任务包，以获得下一个工作。

**讨论**。只要任务数目远超 worker 数目，并且每个任务的开销差别不是很大，就不会出现某些 worker 的工作时间比其他 worker 长很多的情况。在这种情况下，主 / 从算法具有很好的可扩展性。

任务包管理需要全局通信，全局通信开销会影响算法效率。然而，当与任务相关的工作开销远大于任务管理开销时，这将不是问题。在某些情形中，设计者可能需要增加每个任务的大小，以降低访问全局任务包所需要的时间。 144

主 / 从模式不与任何特定的硬件环境捆绑在一起。应用这种模式的程序能够在从集群到 SMP 的所有机器上工作得很好。当然，如果编程环境支持任务包管理，将会非常有意义。

**检测工作是否完成**。编写主 / 从模式程序的一个挑战是正确确定所有任务是否完成。这

个任务既需要高效完成，同时也必须保证所有的工作在 worker 终止之前全部完成。

- 最简单情况下，在 worker 开始工作之前，把所有任务放置在任务包中。然后每个任务依次被 worker 处理，直到任务包为空为止，此时 worker 终止工作。
- 另一种方法是使用队列实现任务包，安排 master 或者一个 worker 检测所期望的终止条件。当检测到终止条件时，创建一个 poison pill，即一个特殊的任务，用于通知 worker 终止工作。把 poison pill 放置到任务包中，且必须保证它能够在获取下一个任务时取出。根据共享任务集的管理方式，可能需要为每一个余下的 worker 创建一个
145 poison pill，以确保所有的 worker 都能收到终止条件。
- 开始时无法确定任务集的问题将导致很严峻的挑战。例如，当 worker 可以向任务包中添加任务时（例如，在分治模式的应用程序中）将发生这种情况。在这种情况下，当 worker 完成一个任务并发现任务包为空时，并不能确定有没有更多工作要做。这是因为另一个仍然活跃的 worker 可能会产生新任务。因此，必须确保任务包为空并且所有的 worker 结束工作。更进一步讲，在基于异步消息传递的系统中，必须确定系统中没有正在传递的消息。因为在这些消息到达后，可能会导致新任务的创建。幸运的是，许多著名算法解决了这个问题。例如，假设任务在概念上被组织为一棵树，其中根是主任务，任务的孩子是它所生成的子任务。当一个任务的所有孩子都终止后，该任务才可以终止。当主任务的所有孩子都终止时，计算将终止。在 [BT89、Mat87、DS80] 中描述了一些终止检测算法。

**变体**。主 / 从模式存在几种变体。因为这种模式以一种非常简单的方式实现了动态负载均衡，所以它非常流行，特别是在易并行问题中（如在易并行模式中所描述的一样）。下面是一些常见的变体。

- master 创建任务后，可转变为一个 worker。当不需要 master 就可以检测到终止条件时（即任务可以根据任务包的状态检测到终止条件），这种技术非常有效。
- 当并发任务可以映射为一个简单循环时，master 可以是隐式的。并且该模式可以实现为动态分配的迭代循环（如 4.4 节所述）。
- 集中式任务队列可能会成为瓶颈，特别是在一个分布式存储器环境中。基于随机工作偷取的方法是一种较优的解决方案 [FLR98]。在该方法中，每一个 PE 维持一个独立的双向任务队列，把新任务放置在本地 PE 任务队列的前端。当一个任务完成时，PE 从本地任务队列的前端移除一个子问题。如果本地任务队列为空，则随机选择另一个 PE，并“偷取”该 PE 任务队列后端的一个子任务。如果这个队列也为空，则随机选择另外一个 PE 继续尝试。这种技术对于基于分治模式的问题特别有效。在这种情形中，队列后端的任务是较早插入的任务，因此代表较大的子问题。这样，这种方法趋向移动较大的子问题，而在创建子问题的 PE 上处理粒度较细的子问题。这种方法有
146 助于负载平衡，并且降低了在较深的递归层中创建小任务的开销。
- 修改主 / 从模式，以提供较好的容错性 [BDK95]。master 需要维持两个队列：一个用于存放仍然需要分配给 worker 的任务，另一个用于存放已经分配给 worker 但未完成的任务。在第一个队列为空后，master 将“未完成”队列中多余的任务分配给它。因此，如果一个 worker 意外终止从而不能完成它的任务，另外一个 worker 将会完成这些未完成的任务。

5. 示例

我们将从一个简单的主 / 从问题开始，提供一个详细示例。该示例在程序的并行实现中使用了主 / 从模式，以生成一个 Mandelbrot 集合。另外，请查看 5.9 节的示例，该部分通过为使用派生 / 聚合模式程序开发一个简单 Java 框架的主 / 从实现，演示了共享队列的使用方式。

**一般解决方案**。主 / 从模式程序的关键是定义存储任务包的数据结构。本节的代码使用任务队列。如图 5-15 所示，我们将这个任务队列实现为共享队列模式的一个实例。

master 进程初始化任务队列，并用整数表示任务；然后使用派生 / 聚合模式创建 worker 进程或线程，并等待它们完成；当这些进程或线程完成时，master 处理它们生成的中间结果。

```
Int const Ntasks // Number of tasks
Int const Nworkers // Number of workers

SharedQueue :: task_queue; // task queue
SharedQueue :: global_results; // queue to hold results

void master()
{
   void worker()

   // Create and initialize shared data structures
   task_queue = new SharedQueue()
   global_results = new SharedQueue()

   for (int i = 0; i < N; i++)
      enqueue(task_queue, i)

   // Create Nworkers threads executing function Worker()
   ForkJoin (Nworkers, Worker)

   consume_the_results (Ntasks)
}
```

图 5-15 主 / 从模式程序的 master 进程。假设存在一个共享地址空间，任务和结果队列对于所有 UE 都可见。在这个简单版本中，master 进程初始化队列，启动 worker，等待所有 worker 完成任务（ForkJoin 命令启动 worker，等待它们完成任务后返回）。当所有 worker 完成任务后，master 处理生成的中间结果，计算完成

如图 5-16 所示，worker 循环处理任务，直到任务队列为空。在循环的每次迭代中，worker 获取任务，完成所指定工作，并将结果存储在全局结果队列中。当任务队列为空时，worker 终止工作。

```
void worker()
   {
     Int :: i
     Result :: res

     while (!empty(task_queue) {
        i = dequeue(task_queue)
        res = do_lots_of_work(i)
        enqueue(global_results, res)
     }
  }
```

图 5-16 主 / 从模式程序的 worker 进程。假设存在一个共享地址空间使 task_queue 和 global_results 对 master 和所有 worker 都可见。worker 循环处理 task_queue 中的任务，当 task_queue 为空时退出

注意，通过使用共享队列的实例，确保了对关键共享变量（task_queue 和 global_results）的安全访问。

Java 程序可使用线程安全队列存储被一组线程执行的 Runnable 对象。这些线程的运行方式和前面描述的 worker 线程类似：从队列中移除一个 Runnable 对象并执行其 run 方法。Java 2 1.5 的 java.util.concurrent 包中的 Executor 接口提供了对主 / 从模式的直接支持。实现该接口的几个类都提供了 execute 方法，该方法将会获取 Runnable 对象并执行。除此之外，这几个类还分别提供了管理实际参与工作的 Thread 对象的几种不同方式。ThreadPoolExecutor 通过使用固定线程池执行命令，从而实现了主 / 从模式。要使用 Executor，程序需要实例化一个实现这个接口的类，这通常通过调用 Executor 类中的一个工厂方法来实现。图 5-17 中的代码构建了一个具有 num_threads 个线程的 ThreadPoolExecutor。这些线程将执行放在一个极大队列中的 Runnable 对象所指定的任务。

```
/*create a ThreadPoolExecutor with an unbounded queue*/
Executor exec = new Executors.newFixedThreadPool(num_threads);
```

图 5-17 实例化并初始化一个 ThreadPoolExecutor

创建 Executor 后，可以将 Runnable 对象（其 run 方法定义了任务的行为）传递给 execute 方法，该方法将会调度该 Runnable 对象的执行。例如，假设 Runnable 对象关联一个可变任务，则对于前面定义的 Executor，exec.execute(task)；语句会将这个任务放置于队列中，该任务最终将被该 Executor 的一个 worker 线程所执行。

主 / 从模式也可用于 SPMD 和 MPI 程序，只是维持全局队列更具有挑战性，但总体算法相同。在 MPI 中使用共享队列的详细描述见第 6 章。

**Mandelbrot 集合的生成。**5.4 节的示例详细描述了 Mandelbrot 集合的生成算法。其基本思想是利用一个二次递归关系式计算复平面上的每一个点，并根据递归关系式在这一点的收敛或者发散率对该点进行着色。复平面上的每一点可独立计算，因此这个问题是易并行问题（参见 4.4 节）。

图 5-18 重新给出了该问题串行版本的伪代码（5.4 节中曾经给出过）。该程序循环处理图像的每一行，对每一行进行计算和显示。

```
Int const Nrows // number of rows in the image
Int const RowSize // number of pixels in a row
Int const M // number of colors in color map

Real :: conv // divergence rate for a pixel
Array of Int :: color_map (M) // pixel color based on Conv rate
Array of Int :: row (RowSize) // Pixels to draw
Array of real :: ranges(2) // ranges in X and Y dimensions

manage_user_input(ranges, color_map) // input ranges, color map
initialize_graphics(RowSize, Nrows, M, ranges, color_map)

for (int i = 0; i<Nrows; i++){

    compute_Row (RowSize, ranges, row)

    graph(i, RowSize, M, color_map, ranges, row)

} // end loop [i] over rows
```

图 5-18 Mandelbrot 集合生成程序串行版本的伪代码

对于同构集群或轻负载的共享内存多处理器计算机，基于 SPMD 模式或循环并行模式的方法是最高效的。然而，对于一个异构集群或一个给众多用户共享的多处理器计算机系统（PE 上的工作负载在任意时间内不可预测），主 / 从模式将更高效。

基于前面描述的高级结构，我们构建了一个主 / 从模式的并行 Mandelbrot 程序。Master 将负责绘制结果。在有些问题中，worker 的运行结果会相互影响。因此，对于 master 来说，直到所有 worker 完成运行才处理中间结果，这一点是非常重要的。然而，在 Mandelbrot 示例中，worker 运行结果相互独立。因此我们分离 `Fork` 和 `Join` 操作，master 获得 worker 的运行结果后，立即绘制它们。在 `Fork` 操作后，master 必须等待结果被放置到 `global-results` 队列中。因为预先知道一行对应一个结果，所以 master 事先知道需要取多少个结果，并且可以简单地用循环迭代数量表示终止条件。当所有结果绘制完成之后，master 在 `Join` 函数处等待所有 worker 完成工作，此时 master 也完成了工作。代码如图 5-19 所示。注意，这段代码与前面描述的普通情形相似。不同之处是，通过分离 `Fork` 和 `Join` 操作，重叠了结果的处理与计算。顾名思义，`Fork` 启动运行指定功能的 UE，而 `Join` 操作使 master 等待所有 worker 彻底终止。有关队列的详情，请查看 5.7 节。

```
Int const Ntasks // number of tasks
Int const Nworkers // number of workers
Int const Nrows // number of rows in the image
Int const RowSize // number of pixels in a row
Int const M // number of colors in color map
typedef Row :: struct of {
    int :: index
    array of int :: pixels (RowSize)
} temp_row;
Array of Int :: color_map (M) // pixel color based on conv rate
Array of Real :: ranges(2) // ranges in X and Y dimensions
SharedQueue of Int :: task_queue; // task queue
SharedQueue of Row :: global_results; // queue to hold results

void master()
{
   void worker();

   manage_user_input(ranges, Color_map) // input ranges, color map
   initialize_graphics(RowSize, Nrows, M, ranges, color_map)

   // Create and initialize shared data structures
   task_queue = new SharedQueue();
   global_results = new SharedQueue();

   for (int i = 0; i < Nrows; i++)
      enqueue(task_queue, i);

   // Create Nworkers threads executing function worker()
   Fork (Nworkers, worker);

   // Wait for results and graph them as they appear
   for (int i = 0; i< Nrows; i++) {
       while (empty(task_queue) { // wait for results
          wait
       }
       temp_row = dequeue(global_results)
       graph(temp_row.index, RowSize, M, color_map, ranges, Row.pixels)
   }

   // Terminate the worker UEs
   Join (Nworkers);
}
```

图 5-19　Mandelbrot 集合生成程序的主 / 从模式并行版本的 master 进程

如图 5-20 所示，worker 的代码比较简单。首先，注意，假设诸如队列和计算参数等是共享变量，因此这两个变量对 master 和 worker 是全局可见的。因为队列在 worker 派生之前由 master 填充，所以终止条件可简单定义为队列为空。每一个 worker 循环获取对应行索引，进行计算，将行索引和计算结果放置到结果队列中，直到队列为空。

```
void worker()
{
  Int i, irow;
  Row temp_row;

  while (!empty(task_queue) {
     irow = dequeue(task_queue);
     compute_Row (RowSize, ranges, irow, temp_row.pixels)
     temp_row.index = irow
     enqueue(global_results, temp_row);
  }
}
```

图 5-20 Mandelbrot 集合生成程序的主 / 从模式并行版本的 worker 进程。设置共享地址空间，使 task_queue、global_results 和 ranges 对 master 和 worker 都可见

**知名应用**。这种模式广泛用于 Linda 编程环境。如 [CG91] 和调查报告 [CGMS94] 所述，Linda 中的元组空间非常适合于那些使用主 / 从模式的程序。

主 / 从模式应用于许多分布式计算环境中，因为这些系统的可用资源是不可预测的。作为搜索地外智慧（SETI）的一部分，SETI@home 项目［SET］使用主 / 从模式借助志愿者的联网计算机下载并分析射电望远镜数据。Calypso 是一种支持 PE 集合动态改变的分布式计算框架 [BDK95]，同样使用主 / 从模式。基因组数据重复检测 [RHB03] 的一个并行算法也使用由 MPI 实现的主 / 从模式，该算法运行在由双处理器 PC 搭建的集群中。

6. 相关模式

147 ~ 151

当循环要使用某种形式的动态调度策略时（例如，在 OpenMP 中使 schedule(dynamic) 子句），这种模式与循环并行模式紧密相关。

派生 / 聚合模式有时也在后台使用主 / 从模式实现。这种模式与使用 TCGMSG[Har91、WSG95、LDSH95] 中的 nextval 函数的算法密切相关。nextval 函数实现一个单调计数器。如果任务包可以映射到一个固定范围内的单调索引上，则可用计数器构建任务包，并且 master 的功能可以用计数器隐式实现。

最后，5.4 节讨论的分子动力学示例中的 owner-computes 过滤器实质上是 master/worker 进程的一种变体。在这样一个算法中，master 所要做的是建立任务包（循环迭代）并将它们分配给 UE，其中任务分配给 UE 的方式由过滤器所定义。因为 UE 实质上可以自己完成任务的分配（通过利用过滤器检测每一个任务），所以不需要显式的 master。

## 5.6 循环并行模式

1. 问题

给定一个串行程序，程序热点为一组计算密集循环，如何将它转换为并行程序？

2. 背景

科学和工程应用的大量程序都基于循环迭代结构。集中于循环来优化这些程序，是对较

老的向量超级计算机的一种传统回溯。将这种方法扩展到现代并行计算机中，得到一种新的并行算法策略，在该算法策略中，并行任务定义为可并行循环的迭代。

对于那些已经具有良好程序的应用来说，围绕可并行循环构造并行算法是非常重要的。在许多情形中，大规模重构一个已有程序以获得并行性能，是不现实的。当程序代码非常复杂、程序算法难以理解时（经常会出现这种情况），这样做是非常重要的。

解决这种问题的方法是构造基于循环的程序，以进行并行计算。当可以获得已有代码，并要将其从串行程序“演化”为并行程序时，所采用的方法是对循环进行一系列转换。理想情况下，对代码的所有改变局限于对循环的转换，这种转换需要消除循环依赖性，但并不改变整个程序的语义（这称为*语义中立转换*）。

并不是所有问题都能够使用这种循环驱动的方法解决。显然，仅当算法的大多数（如果不是所有）计算密集型工作都位于一定数目的不同循环中时，这种方法才有效。更进一步讲，循环迭代必须能作为并行任务很好地工作（也就是说，它们是计算密集的，具有充分的并发性，并且几乎是独立的）。

同时，并不是所有的目标计算机系统都适合该类型的并行程序。如果不能重构代码，以创建高效的分布式数据结构，则对共享地址提供某种程度的支持是至关重要的，但也是最普 152
通的情形。最后，Amdahl 定理及其尽量最小化程序串行部分的需求，通常意味着基于循环的方法仅在那些具有较小数目 PE 的计算机系统中能够取得较高效率。

尽管存在这些限制，但这类并行算法的发展很迅速。这是因为基于循环的算法是高性能计算的传统方法，在新程序中仍然占据支配地位。目前存在大量基于循环的程序需要移植到现代并行计算机中。创建 OpenMP API 的主要目的是为了支持这些循环驱动问题的并行化。这些算法的可扩展性非常差，但可以接受，因为具有两个或四个处理器的机器远比具有数十或数百个处理器的机器多得多。

*这种模式与运行在共享内存计算机上的程序，以及与那些使用任务并行模式和几何分解模式的问题密切相关。*

*3. 面临的问题*

- **串行等价性**。如果程序单线程或多线程执行，能够产生一致的结果（除舍入误差外），则认为该程序具有*串行等价性*。串行等价性代码比较容易编写和维护，并且一份程序源代码能同时在串行机器和并行机器上工作。
- **增量并行性（或重构）**。当对一个已有程序进行并行化时，如果并行化为一系列增量转换，即一次转换一个循环；并且转换不“破坏”程序，允许在每次转换后进行测试，则得到一个正确的并行程序会相对容易。
- **内存使用**。良好的性能需要数据访问模式（在循环中隐含）能与系统的内存层次很好地融合在一起。这个问题与前两个问题不一样，程序员需要对循环进行大规模重构。

*4. 解决方案*

该模式与 OpenMP 所隐含的并行编程风格密切相关，基本方法由下面几个步骤组成。

- **定位性能瓶颈**。寻找计算最密集的循环，主要方式为：剖析代码；了解每一个子问题的性能需求；使用程序性能分析工具。这些循环在典型数据集上的总运行时间将最终限制并行程序的可扩展性（参考 Amdahl 定理）。 153

- **消除循环依赖性**。寻找迭代间或读/写访问间的依赖性，进而转换代码以消除或者缓解这些依赖性，确保循环的每次迭代必须近似独立。4.4 节讨论了如何寻找和消除依赖性，5.8 节也讨论了如何利用同步结构保护这些依赖性。
- **并行化循环**。在 UE 间划分循环迭代。为保证串行等价性，程序需要使用语义中立指令，如 OpenMP 提供的这种指令（见附录 A）。理想情况下，应当一次只处理一个循环，并进行细致的检查和测试，以确保没有引入竞争条件或其他错误。
- **优化循环调度**。迭代必须能够在程序执行时被 UE 调度，以确保负载平衡。正确的调度策略依赖于对问题地清晰理解，也需要频繁地试验，以找到最优的调度策略。

仅在循环迭代的计算时间足够长，能够补偿并行循环开销时，这种方法才有效。每个循环的迭代数量也非常重要，如果每个 UE 可以分配多个迭代将增加调度灵活性。在某些情况下，可能需要进行代码转换以解决这些问题。

通常使用的两种代码转换方式如下。

- **循环合并**。如果问题由一系列循环组成，并且这些循环具有一致的循环限制，那么这些循环通常可以合并为一个具有更复杂迭代的循环，如图 5-21 所示。

```
#define N 20
#define Npoints 512

void FFT(); // a function to apply an FFT
void invFFT(); // a function to apply an inverse FFT
void filter(); // a frequency space filter
void setH(); // Set values of filter, H

int main() {
   int i, j;
   double A[Npoints], B[Npoints], C[Npoints], H[Npoints];

   setH(Npoints, H);

   // do a bunch of work resulting in values for A and C

   // method one: distinct loops to compute A and C
   for(i=0; i<N; i++){
      FFT (Npoints, A, B); // B = transformed A
      filter(Npoints, B, H); // B = B filtered with H
      invFFT(Npoints, B, A); // A = inv transformed B
   }
   for(i=0; i<N; i++){
      FFT (Npoints, C, B); // B = transformed C
      filter(Npoints, B, H); // B = B filtered with H
      invFFT(Npoints, B, C); // C = inv transformed B
   }

   // method two: the above pair of loops combined into
   // a single loop
   for(i=0; i<N; i++){
      FFT (Npoints, A, B); // B = transformed A
      filter(Npoints, B, H); // B = B filtered with H
      invFFT(Npoints, B, A); // A = inv transformed B
      FFT (Npoints, C, B); // B = transformed C
      filter(Npoints, B, H); // B = B filtered with H
      invFFT(Npoints, B, C); // C = inv transformed B
   }
   return 0;
}
```

图 5-21 循环合并的代码片段，为了增加每次迭代的工作量

- **合并嵌套循环**。嵌套循环通常能够组合为一个具有较大组合迭代次数的循环，如图 5-22 所示。较大次数的迭代有助于减轻并行循环开销，这主要是因为增加了循环并发性，能够更好地利用较大数目的 UE；为迭代调度到 UE 上的方式提供了额外选项。

```
#define N 20
#define M 10

extern double work(); // a time-consuming function

int main() {

   int i, j, ij;
   double A[N][M];

   // method one: nested loops

   for(j=0; j<N; j++){
      for(i=0; i<M; i++){
         A[i][j] = work(i,j);
      }
   }

   // method two: the above pair of nested loops combined into
   // a single loop.

   for(ij=0; ij<N*M; ij++){
      j = ij/N;
      i = ij%M;

      A[i][j] = work(i,j);
   }

   // method three: the above loop parallelized with OpenMP.
   // The omp pragma creates a team of threads and maps
   // loop iterations onto them. The private clause
   // tells each thread to maintain local copies of ij, j, and i.

#pragma omp parallel for private(ij, j, i)
   for(ij=0; ij<N*M; ij++){
      j = ij/N;
      i = ij%M;

      A[i][j] = work(i,j);
   }
   return 0;
}
```

图 5-22 合并嵌套循环的程序片段，为了产生一个具有较大迭代次数的循环

OpenMP 语言使用 `omp parallel for` 指令，可很容易完成循环的并行化。这条指令告诉编译器创建一组线程（共享内存环境中的 UE），并在这些线程间对循环迭代进行分配。图 5-22 中的最后一段代码是 OpenMP 的循环并行化示例。第 6 章从一个较高层次描述这个指令。语义细节请参考附录 A。

注意，在图 5-22 中，必须为每个线程创建索引 `i` 和 `j` 的副本。使用这种模式最常见的错误是忽略了“私有”关键变量。如果 `i` 和 `j` 是共享变量，那么不同 UE 对 `i` 和 `j` 的更新可能会引发冲突，并导致不可预测的结果（即程序中含有竞争条件）。编译器通常不检测这些错误，因此程序员必须十分小心，确保避免类似错误的产生。

应用这种模式的关键是使用语义中立指令修改，以产生串行等价代码。语义中立修改不

改变单线程程序的语义。当正确使用前面描述的循环合并和嵌套循环合并技术时，对代码的修改都是语义中立修改。另外，OpenMP 中的大多数指令都是语义中立的。这就意味着，添加 OpenMP 指令并单线程运行，运行结果和没有 OpenMP 指令的原始程序的运行结果相同。

两个语义等价（当单线程运行时）的程序不一定是串行等价的。无论是单线程运行还是多线程运行，串行等价都意味着程序将给出相同的运行结果（因改变浮点操作的顺序而产生的舍入误差除外）。事实上，消除循环依赖性的（语义中立）动力来自于期望将一个串行不等价的程序改变为一个串行等价的程序。当进行程序转换以提高性能时，即使转换是语义中立的，也要仔细注意没有丧失串行等价性。

154
~
156

当代码指定线程 ID 或者线程数目时，定义串行等价程序将更加困难。访问线程 ID 和线程数目的算法趋向于访问特定的线程或者特定的线程数目。当目标是一个串行等价程序时，这是一种非常危险的情况。

当算法依赖于线程 ID 时，程序员正在应用 SPMD 模式。这可能让人困惑。SPMD 程序可以是基于循环的。事实上，SPMD 模式中的许多示例都是基于循环的算法。但它们不是*循环并行*模式的示例，因为它们演示了一个 SPMD 程序的特性——即使用了 UE 的 ID 来引导算法。

最后，假设当使用这种编程模式时，我们有一个基于指令的系统（例如 OpenMP）。如果没有这种基于指令的编程环境，应用这种模式虽然可行但非常困难。在面向对象的设计中，通过灵活地使用具有并行 iterator 的匿名类来使用*循环并行*模式。因为并行性隐含在 iterator 中，所以可以满足串行等价性条件。

**性能考虑**。使用这种模式的几乎每个应用程序中，特别是使用 OpenMP 时，通常假设程序运行在一台具有多个 PE 的计算机上。这些 PE 共享地址空间，并假设这个地址空间对每个存储元素提供等时访问。

遗憾的是，通常这种情况很难满足。现代计算机的内存组织具有层次性，PE 具有缓存，内存模块与 PE 的子集封装在一起。虽然在设计共享内存多处理器计算机时，努力使它们能够像对称多处理器（SMP）计算机一样工作。但事实上所有共享内存计算机都展示了系统中某种程度的不一致内存访问时间。在许多情况下，这些并不是优先考虑的因素，我们可以忽略程序的内存访问模式与目标计算机系统内存层次的匹配程度。然而，在有些情况下，特别是对于较大的共享内存机器，必须根据内存层次的需要显式组织程序。最常用的技巧是确保关键数据结构在初始化期间的数据访问模式与后续计算使用这些数据结构的数据访问模式相匹配。在 [Mat03、NA01] 中以及本节后面，更详细地讨论了这一点。

另一个性能问题是伪共享。当不被 UE 所共享的一些变量恰好驻留在相同的缓存行中时，就发生这个问题。因此，尽管程序在语义上是独立的，但每个 UE 的每次访问都需要在 UE 间移动一个缓存行，这可能会带来巨大的开销。因为当这些所谓的独立变量被更新时，缓存行会反复在 UE 间移动。图 5-23 中给出了这样一个存在伪共享的程序片段。代码中有一个嵌套循环，外层循环的迭代次数较少，映射到 UE 的数目（在此假设这个迭代次数为 4）。内层循环的迭代次数较多且比较耗时。假设对于外层循环的每次迭代，内层循环迭代次数大致相等，

157

则这个循环将高效地并行化。但是内层循环对数组 A 元素的每次更新都需要 UE 请求对应的缓存行。尽管数组 A 的元素在 UE 间是完全独立的，但是它们很可能位于同一缓存行中。因此，内层循环的每次迭代将产生一次昂贵的缓存行“无效移动”操作。这个问题不仅会降低

并行加速比，而且当PE数目增多时，并行程序还会变慢。解决方案是在每一个线程中创建一个私有临时变量，在内层循环中作为中间结果。虽然此时仍然存在伪共享，但仅针对迭代次数较少的外部循环，带来的性能影响是可以忽略的。

```
#include <omp.h>
#define N 4 // Assume this equals the number of UEs
#define M 1000

extern double work(int, int); // a time-consuming function

int main() {

   int i, j;
   double A[N] = {0.0}; // Initialize the array to zero

// method one: a loop with false sharing from A since the elements
// of A are likely to reside in the same cache line.

#pragma omp parallel for private(j,i)
   for(j=0; j<N; j++){
      for(i=0; i<M; i++){
          A[j] += work(i,j);
      }
   }

// method two: remove the false sharing by using a temporary
// private variable in the innermost loop

   double temp;

#pragma omp parallel for private(j,i, temp)
   for(j=0; j<N; j++){
      temp = 0.0;
      for(i=0; i<M; i++){
          temp += work(i,j);
      }
      A[j] += temp;
   }
   return 0;
}
```

图 5-23 伪共享程序片段。小数组A存储在一个或两个缓存行中。当UE在内层循环中访问A时，它们将从其他UE处取回该缓存行的所有权。缓存行的这种前后移动将降低系统性能。解决方案是在内层循环中使用一个临时变量 158

5. 示例

该模式的示例如下：

- 数值积分，使用梯形规则估算一个定积分的值；
- 分子动力学，非化学键能量计算；
- Mandelbrot集合计算；
- 网格计算。

这些示例在本书的其他章节已经详细描述过了。本节着重讨论循环，以及它们并行化的方式。

**数值积分**。考虑使用式（5-5）估算 π 值的问题：

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \qquad (5\text{-}5)$$

我们使用梯形规则来求解这个积分。其基本思想是用一系列矩形填充一条曲线之下的区域。当矩形宽度接近于0时，矩形面积总和接近于积分值。

图5-24显示了在单处理器上进行这种计算的串行程序。为简单起见，我们将积分中的迭代次数固定为1 000 000。变量sum初始化为0，步长通过x（在这种情形中，x为1.0）除以总迭代次数计算得到。每一个矩形的面积为宽（步长）乘以高（被积函数在区间的中心值）。因为宽是一个常量，所以我们将其放在迭代循环之外，并使用步长（step）乘以矩形高度的总和，以获得定积分的估算值。

```
#include <stdio.h>
#include <math.h>

int main () {
   int i;
   int num_steps = 1000000;
   double x, pi, step, sum = 0.0;

   step = 1.0/(double) num_steps;

   for (i=0;i< num_steps; i++)
   {
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
   printf("pi %lf\n",pi);
   return 0;
}
```

图5-24 利用梯形规则估算 π 值的串行代码

使用*循环并行模式*构建该程序的并行版本比较简单。程序仅包含一个循环，因此检查阶段非常简单。为保持循环迭代的独立性，我们认为：变量x的值对每次迭代来说都是局部的，因此这个变量可定义为线程的局部变量或私有变量；sum更新需定义一次归约操作。OpenMP的API支持归约操作。除添加#include<omp.h>[⊖]语句外，只需在for循环之前再添加如下一行代码，就可以构建该程序的一个并行版本：

```
#pragma omp parallel for private(x) reduction(+:sum)
```

这条pragma语句告诉OpenMP编译器：①创建一个线程组；②为每个线程创建x和sum的一个私有副本；③将sum初始化为0（对于加法为单位操作数）；④将循环迭代映射到线程中；⑤将sum的局部结果归约为一个全局结果；⑥将所有的并行线程聚合到主线程。第6章和（附录A）对每个步骤进行了详细描述。对于非OpenMP编译器，这条pragma语句将被忽略，因此对程序的行为不产生任何影响。

**分子动力学**。本书通篇将使用分子动力学算法作为一个递归示例介绍这种仿真。分子动力学仿真了一个大型分子系统的运动。该算法使用一个显式时间步方法，即计算出每个原子在每个时间步的作用力，然后使用经典力学中的标准算法来计算作用力是如何改变原子运动的。

⊖ 这个头文件文件定义了OpenMP所使用的函数原型和不透明的数据类型。

该应用的核心算法（包括伪代码），在 3.1.3 节和 5.4 节中都介绍过。该程序由一系列作用在原子上的计算昂贵的循环组成（在分子系统范围内）。这些循环嵌套在一个作用在时间上的顶层循环中。

作用在时间上的循环不能并行化，因为第 $t-1$ 步的坐标和速度是第 $t$ 步的起点。但是，作用在原子上的每个循环都能够并行化。其中要解决的关键问题是非化学键能量的计算。该计算的代码如图 5-25 所示。与 5.4 节的示例中所使用的方法不同，假设程序和它的数据结构与串行代码相同。

```
function non_bonded_forces (N, Atoms, neighbors, Forces)

    Int N // number of atoms

    Array of Real :: atoms (3,N) //3D coordinates
    Array of Real :: forces (3,N) //force in each dimension
    Array of List :: neighbors(N) //atoms in cutoff volume
    Real :: forceX, forceY, forceZ

    loop [i] over atoms

        loop [j] over neighbors(i)
            forceX = non_bond_force(atoms(1,i), atoms(1,j))
            forceY = non_bond_force(atoms(2,i), atoms(2,j))
            forceZ = non_bond_force(atoms(3,i), atoms(3,j))
            force(1,i) += forceX; force(1,j) -= forceX;
            force(2,i) += forceY; force(2,j) -= forceY;
            force(3,i) += forceZ; force(3,j) -= forceZ;
        end loop [j]

    end loop [i]
end function non_bonded_forces
```

图 5-25 一段典型的分子动力学并行代码中非化学键作用力计算的伪代码。该代码与图 4-4 所示的串行函数版本几乎一致

我们将并行化作用于原子的循环。注意，变量 forceX、forceY 和 forceZ 是迭代内部使用的临时变量。我们将在每个 UE 上创建这些变量的副本。force 数组的更新是归约操作。这些函数的并行化需要在作用于原子的循环之前添加一条指令：

```
#pragma omp parallel for private(j, forceX, forceY, forceZ) \
                    reduction (+ : force)
```

每个原子的工作量是不可预测的，这依赖于该原子邻近区域中的原子数目。尽管编译器可以假定一种高效的调度方式，但在这个问题中，最好尝试不同的调度方式，以寻找最优的工作方式。因为每个原子的工作量是不可预测的，所以应当使用 OpenMP 的一种动态调度策略（在附录 A 中，有关于这些策略的描述）。这需要添加一个 schedule 子句。这是并行化这个程序的最后一条 pragma 语句：

```
#pragma omp parallel for private(j, forceX, forceY, forceZ) \
                reduction (+ : force) schedule (dynamic,10)
```

159 ~ 161

这个 schedule 子句告诉编译器将循环迭代分成大小为 10 的任务块，并将它们动态地分配给 UE。任务块的大小是任意的，需根据动态调度开销和负载均衡的效率来综合选定。

C/C++ 版的 OpenMP 2.0 不支持数组归约操作，因此需要显式地进行归约。这个工作非常简单（如图 5-11 所示）。将来的 OpenMP 版本将纠正这个缺陷，在所有支持 OpenMP 的语

言中都提供对数组归约操作的支持。

本书中的所有分子动力学程序将使用相同方法对非化学键作用力计算进行并行化。性能和可扩展性成为构建该程序 SPMD 版本的主要障碍。这是因为每次遇到并行指令，就需要创建一个新的线程组。大多数 OpenMP 实现使用线程池，而不是为每个并行区域实际创建一个新的线程组，这将最小化线程的创建和销毁开销。但是，这种并行化计算方法仍然会带来严重的额外开销，而且缓存中数据的重用性较差。原则上，UE 上的每个循环可以通过一种不同的模式访问原子。但这会降低 UE 高效利用已经位于缓存中数据的能力。

即使存在这些缺点，当面向一个小型共享内存计算机系统进行算法并行化时，通常仍使用这种方法 [BBE+99]。例如，可在一个集群中使用 SPMD 版本的分子动力学程序，然后使用 OpenMP，通过在双处理器或多微处理器上同时使用多线程来获得额外性能 [MPS02]。

**Mandelbrot 集合计算**。考虑著名的 Mandelbrot 集合 [Dou86]。4.4 节和 5.4 节对该算法以及并行化进行了讨论。该算法基于式（5-6）中的二次递归关系式对每一个像素进行着色。

$$Z_{n+1} = Z_n^2 + C \tag{5-6}$$

式中，$C$ 和 $Z$ 是复数，递归从 $Z_0 = C$ 开始。图像在垂直轴（-1.5 ~ 1.5）上绘制 $C$ 的虚部，在水平轴（-1 ~ 2）上绘制 $C$ 的实部。如果递归关系收敛于一个稳定值，则每个像素的颜色都是黑色；否则，将根据递归关系的发散率对像素进行着色。

图 5-26 展示了该算法串行版本的伪代码。程序的核心计算隐藏在 `compute_row()` 例程中。因为该例程的细节对于理解并行算法来说并不重要，所以在这里将不讨论。对于一行中的每个点，需要进行如下处理。

- 每个像素对应于二次递归关系式中 $C$ 的一个值。我们基于输入 range 和像素的索引计算这个值。
- 计算递归中的项，并基于它是否收敛于一个固定值或者发散率来设置像素值。如果它发散，就基于发散率设置像素值。

```
Int const Nrows // number of rows in the image
Int const RowSize // number of pixels in a row
Int const M // number of colors in color map

Real :: conv // divergence rate for a pixel
Array of Int :: color_map (M) // pixel color based on conv rate
Array of Int :: row (RowSize) // Pixels to draw
Array of Real :: ranges(2) // ranges in X and Y dimensions

manage_user_input(ranges, color_map) // input ranges, color map
initialize_graphics(RowSize, Nrows, M, ranges, color_map)

for (int i = 0; i<Nrows; i++){

    compute_Row (RowSize, ranges, row)

    graph(i, RowSize, M, color_map, ranges, row)

} // end loop [i] over rows
```

图 5-26 Mandelbrot 集合生成程序串行版本的伪代码

计算完成后，绘制所有行，从而绘制出著名的 Mandelbrot 集合图像。根据发散率到一个

颜色表的映射来确定像素颜色。

使用*循环并行*模式构建该程序的并行版本比较简单。作用在行上的循环迭代相互独立，只需要确保每一个线程只处理属于自己的行。利用一个 pragma 语句来完成这个工作：

```
#pragma omp parallel for private(row)
```

任务调度可能有点麻烦，因为每一行的工作量依赖于点的发散程度，所以差别较大。程序员应当尝试几种不同的调度策略，但是周期调度策略很可能会实现高效的负载均衡。在周期调度策略中，循环迭代像分发纸牌一样调度运行。通过将循环迭代交错地分配到一组线程中，很可能会实现较好的负载均衡。因为这种调度策略是静态的，所以所带来的额外开销也较小。

162
~
163

```
#pragma omp parallel for private(Row) schedule(static, 1)
```

关于 `schedule` 子句和并行程序员所能够使用的不同选项的详细信息，请参考附录 A。

注意，我们假设程序中所使用的制图包是线程安全的，即当多个线程同时调用这个库时，不会产生任何问题。对于标准 I/O 库，OpenMP 规范要求它具有线程安全性，但是对其他库没有这样的要求。因此，需要将 `gxaph` 函数放置在一个临界区内以进行多线程保护：

```
#pragma critical
      graph(i, RowSize, M, color_map, ranges, row)
```

第 6 章和附录 A 详细地描述这种结构。这种方法虽然能够很好地工作，但如果在相同的时间内，很多行同时进行计算，并且所有线程都尝试绘制它们的行，则这种方法具有严峻的性能问题。

**网格计算**。考虑一种求解 1D 热扩散公式的简单网格计算。4.6 节详细描述了该问题以及基于 OpenMP 的解决方案。图 5-27 中重新给出了这个解决方案。

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define NX 100
#define LEFTVAL 1.0
#define RIGHTVAL 10.0
#define NSTEPS 10000

void initialize(double uk[], double ukp1[]) {
    uk[0] = LEFTVAL; uk[NX-1] = RIGHTVAL;
    for (int i = 1; i < NX-1; ++i)
        uk[i] = 0.0;
    for (int i = 0; i < NX; ++i)
        ukp1[i] = uk[i];
}

void printValues(double uk[], int step) { /* NOT SHOWN */ }

int main(void) {
    /* pointers to arrays for two iterations of algorithm */
    double *uk = malloc(sizeof(double) * NX);
    double *ukp1 = malloc(sizeof(double) * NX);
    double *temp;
    int i,k;
```

图 5-27 基于 OpenMP 的并行热扩散程序。4.6 节的示例介绍过这个程序

```
    double dx = 1.0/NX; double dt = 0.5*dx*dx;

  #pragma omp parallel private (k, i)
    {
       initialize(uk, ukp1);

       for (k = 0; k < NSTEPS; ++k) {
          #pragma omp for schedule(static)
          for (i = 1; i < NX-1; ++i) {
            ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
          }
          /* "copy" ukp1 to uk by swapping pointers */
          #pragma omp single
          {temp = ukp1; ukp1 = uk; uk = temp;}
       }
    }
    return 0;
}
```

图 5-27 （续）

这个程序在大多数共享内存计算机上能很好地工作。然而，如果仔细分析程序性能，将会发现两个问题。首先，保护共享指针交换的 single 指令增加了一个额外的 barrier 操作，极大增加了同步开销。其次，在 NUMA 计算机上，该算法的访存开销可能会很高，因为我们没有将数组放在处理它们的 PE 附近。

图 5-28 所示的代码解决了这两个问题。为去掉 single 指令，可以通过使用 private 语句使每个线程都具有自己的 uk 和 ukpl 指针的副本。然而，为了更加灵活，需要指向由网格值所组成的共享数组的 uk 和 ukpl 指针的新私有副本。这可以通过使用 firstprivate 子句来实现，该语句应用于创建线程组的 parallel 指令。

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define NX 100
#define LEFTVAL 1.0
#define RIGHTVAL 10.0
#define NSTEPS 10000

void initialize(double uk[], double ukp1[]) {
    int i;
    uk[0] = LEFTVAL; uk[NX-1] = RIGHTVAL;
    ukp1[NX-1] = 0.0;
#pragma omp for schedule(static)
    for (i = 1; i < NX-1; ++i){
        uk[i] = 0.0;
        ukp1[i] = 0.0;
    }
}

void printValues(double uk[], int step) { /* NOT SHOWN */ }

int main(void) {
    /* pointers to arrays for two iterations of algorithm */
    double *uk = malloc(sizeof(double) * NX);
    double *ukp1 = malloc(sizeof(double) * NX);
    double *temp;
    int i,k;
```

图 5-28 基于 OpenMP 的并行热扩散程序，降低了线程管理开销，且内存管理更适于 NUMA 计算机

```
    double dx = 1.0/NX; double dt = 0.5*dx*dx;

#pragma omp parallel private (k, i, temp) firstprivate(uk, ukp1)
    {
       initialize(uk, ukp1);

       for (k = 0; k < NSTEPS; ++k) {
          #pragma omp for schedule(static)
          for (i = 1; i < NX-1; ++i) {
            ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
          }
          /* "copy" ukp1 to uk by swapping pointers */
          temp = ukp1; ukp1 = uk; uk = temp;
       }
    }
   return 0;
}
```

图 5-28 （续）

我们所解决的另外一个性能问题：最小化访存开销，它所使用的方法更微妙。如前所述，为降低访存量，最重要的是数据存储位置尽可能接近处理它们的 PE。在 NUMA 计算机上，这对应于确保将内存页面分配给将要处理这些页面中数据的 PE。最常用的 NUMA 页面布局算法是“first touch”算法。其基本思想是：当 PE 第一次访问某个内存区域时，将包含该内存区域的页面将分配给它。OpenMP 程序经常使用的一种技术是使用与后续计算相同的循环调度策略来并行地初始化数据。

首先稍微改变初始化过程使得初始化循环与计算循环一致；然后在初始化循环中使用与计算循环相同的并行化指令。虽然这并不能保证这是内存页面到 PE 的最优映射，但是非常容易实现，并且在许多情况下，它非常接近于最优解决方案。

**知名应用**。OpenMP 程序员经常使用*循环并行*模式。在北美（Wompat：OpenMP 应用和工具的研讨会）、欧洲（WEOMP：OpenMP 欧洲研讨会）和日本（WOMPEI：OpenMP 经验和实现研讨会）每年都举行研讨会，讨论 OpenMP 及其用法。可以很方便地获得这些研讨会的论文集 [VJKT00、Sci03、EV01]，这些论文集包含*循环并行*模式的很多示例。

164 ~ 166

OpenMP 的大多数工作被限定在共享内存多处理器计算机上，用于解决那些能够在近乎平面的内存层次上就可以很好工作的问题。然而，为了将 OpenMP 应用程序扩展到更复杂的存储层次中，包括 NUMA 机器 [NA01、SSGF00]，甚至是集群 [HLCZ99、SHTS01]，人们已经做了很多工作。

6. *相关模式*

循环集合驱动并行是一个普遍的概念，并用于多种模式中。特别是许多使用 SPMD 模式的问题是基于循环的。但是它们需要使用 UE 的 ID 来并行化循环，因此无法很好地映射到这种模式。另外，一些使用 SPMD 模式的问题在循环间通常会包含某种程度的并行逻辑，这使得它们能够减少程序的串行部分。这也是 SPMD 程序比使用*循环并行*模式的程序更具可扩展性的原因之一。

为共享内存计算机所开发的算法（一般使用*任务并行*模式或*几何分解*模式）通常使用*循环并行*模式。

## 5.7 派生 / 聚合模式

1. 问题

在某些程序中，并发任务数量随程序执行而发生变化，这些任务相关的行为使得简单控制结构（如并行循环）无法实现良好的并行化。如何围绕复杂动态任务集构造并行程序呢？

2. 背景

某些问题的算法使用一种通用和动态的并行控制结构。当程序继续执行时，任务动态创建（即派生），然后聚合终止（即聚合）。在大多数情况下，任务间关系很简单，动态任务创建可以通过并行循环处理（如 5.6 节所述），或者通过任务队列处理（如 5.5 节所述）。然而，在某些情况下，必须通过任务的管理方式来捕获算法中任务间的相互关系。如：递归产生的任务结构，非常不规则的连接任务（connected task）集，以及那些不同函数被映射到不同并发任务的问题。在每一个示例中，任务首先被派生，然后聚合到父任务（即执行 `fork` 操作的任务）和由相同父任务创建的其他任务上。这些问题是利用派生 / 聚合（Fork/Join）模式解决的。

以使用分治模式设计的算法作为示例。当程序执行时，问题被划分为多个子问题，然后
167 递归创建（或者派生）新任务来并发执行这些子问题；这些任务可能会被进一步划分。当处理一个特定划分的所有任务终止并且聚合到父任务时，父任务再继续运行。

这种模式与在共享存储器计算机上运行的 Java 程序和使用分治模式与递归数据模式的问题特别相关。当 OpenMP 环境支持嵌套并行时，也可以高效地使用这种模式。

3. 面临的问题

- 算法隐含了任务间的相互关系。在有些问题中，任务间存在需要动态创建和终止的复杂（或递归）关系。尽管这些关系可以映射到熟悉的控制结构上，但在许多情形中，如果任务结构与 UE 结构相近，则更易于理解设计。
- 任务到 UE 一对一映射是很自然的，但必须要和系统中可用的 UE 数目进行平衡。
- UE 的创建和销毁具有很大的开销。为使这两个操作不会严重影响程序的整体性能，可能需要重新构造算法。

4. 解决方案

当使用派生 / 聚合模式时，任务以不同方法映射到 UE 上。我们将讨论两种方法：直接映射，每个 UE 处理一个任务；间接映射，一个 UE 池处理一个任务集。

**任务 /UE 直接映射**。最简单的情形是将每一个子任务映射到不同 UE 上。当派生新的子任务时，创建新的 UE 处理它们，这将建立对应的任务和 UE 集合。在许多情形中，存在一个同步点，主任务在该点处等待子任务的完成，这称为聚合。UE 完成对应的子任务后将被销毁。我们将在后面提供这种方法的一个示例（使用 Java 编程语言）。

派生 / 聚合模式应用的直接任务 /UE 映射法是 OpenMP 的标准编程模型。程序开始运行时只有一个线程（主线程），随后在并行区域处派生出一组线程。这些线程共享地址空间并在并行区域的结尾处聚合在一起。然后初始主线程继续执行，直到程序结束或直到下一个并行
168 区域[⊖]。这种结构是 5.6 节所描述的 OpenMP 并行循环构建的实现基础。

⊖ 通常，OpenMP 程序中的嵌套并行区域也采用这种直接映射法。这种方法已经成功应用于 [AML+99]。但 OpenMP 规范要求 OpenMP 实现要“串行化”嵌套并行区域（也就说，使用大小为 1 的线程组执行它们）。因此，OpenMP 程序不能依赖嵌套并行区域派生额外线程。当程序员使用 OpemMP 编写最简单的派生 / 聚合程序时一定要谨慎。

**任务/UE间接映射**。并行程序中，线程与进程的创建和销毁开销非常大。如果一个程序含有多个派生和聚合操作，则需要多次构建和销毁UE，这势必会影响程序效率。另外，如果UE数目多于PE，则上下文切换所带来的开销是程序所不能接受的。

在这种情况下，最好使用线程池来实现派生/聚合模式，以避免UE的动态创建。其基本思想是：在第一个派生操作之前，创建一个与PE数目相同的静态UE集，然后使用任务队列将任务动态映射到UE上。UE本身没有重复地创建和销毁，只是实现了和动态创建的任务的简单映射。尽管这种方法实现起来很复杂，但是能在高效的程序中实现很好的负载均衡。我们将在本节的示例中讨论一个使用这种方法的Java程序。

OpenMP对使用这种间接映射方法的最佳实践存在争论[Mat031]。最终赢得人们信任的是一种基于OpenMP工作共享（workshare）结构的方法。这种结构称为taskqueue [SHPT00]。该方法定义了两种新数据结构：taskqueue和task。程序员使用taskqueue来创建任务队列。在taskqueue结构中，task结构定义了将要被封装到任务中并放置到任务队列的代码块。线程组（通常由一个并行构造所创建）扮演了线程池的角色，从队列中取出任务并执行它们，直到队列为空。

与OpenMP并行区域不同，taskqueue可以嵌套并产生层次任务队列。线程使用任务偷取算法来处理这些任务队列，以保持在所有任务队列为空之前，所有线程都繁忙。已经证明这种方法能够很好地工作[SHPT00]，很可能在将来的OpenMP规范中被采用。

5. 示例

下面将分别采用直接映射和间接映射方法实现归并排序算法。间接映射法使用了一个Java包：FJTasks [Lea00b]。5.9节的示例中开发了一种相似的但更简单的框架。

**使用直接映射的合并排序**。

如图5-29所示，考虑一个实现合并排序的简单方法。该方法对数组在[lo,hi)（包含Lo，不包含hi）之间的元素进行排序，通过调用sort(A,0,A.length)就可以排序整个数组A。 169

```
static void sort(final int[] A,final int lo, final int hi)
{ int n = hi - lo;

  //if not large enough to do in parallel, sort sequentially
  if (n <= THRESHOLD){ Arrays.sort(A,lo,hi); return; }
  else
  { //split array
    final int pivot = (hi+lo)/2;

    //create and start new thread to sort lower half
    Thread t = new Thread()
    { public void run()
      { sort(A, lo, pivot); }
    };
    t.start();

    //sort upper half in current thread
    sort(A,pivot,hi);

    //wait for other thread
    try{t.join();}
    catch (InterruptedException e){Thread.dumpStack();}
```

图5-29 归并排序，其中每个任务对应于一个线程

```
      //merge sorted arrays
      int[] ws = new int[n];
      System.arraycopy(A,lo,ws,0,n);
      int wpivot = pivot - lo;
      int wlo = 0;
      int whi = wpivot;
      for (int i = lo; i != hi; i++)
      { if((wlo < wpivot) && (whi >= n || ws[wlo] <= ws[whi]))
        { A[i] = ws[wlo++]; }
        else { A[i] = ws[whi++]; }
      }
    }
  }
```

图 5-29 （续）

该方法第一步计算待排序的数组长度。如果问题规模太小，以至于不值得采用并行方法进行排序，则使用串行排序算法（在这个示例中，使用 java.util 包中的 Arrays 类所提供的快速排序算法）。在并行算法中，首先计算一个支点（pivot）划分待排序数组。然后派生出一个新线程对数组后半部分进行排序，父线程对数组前半部分进行排序。新任务由 Thread 类的一个匿名内部子类的 run 方法所指定。新线程完成排序后即终止运行，父线程完成排序后执行聚合操作等待子线程运行完成。最后父线程将两个已经排序的数组段合并在一起。

这种简单方法非常适用于规则问题。在这些问题中可以很容易地确定某些合理的阈值。合理选择阈值至关重要：如果阈值太小，会产生过多 UE 的开启和销毁操作，可能会导致并行程序比串行程序还要慢。如果阈值太大，则无法完全开发潜在的并行性。

**使用间接映射的合并排序**。这个示例使用了 FJTask 框架，该框架是公共域包 EDU.Oswego.CS.dl.util.concurrent [Lea00b] 的一部分[⊖]。该框架不是创建一个新线程而是创建 FJTask（或其子类）的一个实例来执行每个任务。该包将 FJTask 对象动态映射到一个静态线程集上运行。尽管 FJTask 对象的通用性比 Thread 差，但它比 Thread 更轻，创建和销毁开销更低。图 5-30 和图 5-31 演示了如何修改合并排序示例，以使用 FJTask 对象取代 Java 线程。所需要的类由包 EDU.Oswego.CS.dl.util.concurrent 引入。如图 5-30 所示，在开始实例化任何 FJTask 对象之前，必须首先实例化一个 FJTaskRunnerGroup 对象，该语句将线程数目（组大小）作为参数创建构成线程池的一组线程。一旦被实例化，主任务就会被 FJTaskRunnerGroup 类的 invoke 方法所调用。

```
int groupSize = 4; //number of threads
FJTaskRunnerGroup group = new FJTaskRunnerGroup(groupSize);
group.invoke(new FJTask()
      { public void run()
        { synchronized(this)
          { sort(A,0, A.length); }

        }
      });
```

图 5-30 实例化 FJTaskRunnerGroup 并调用主任务

⊖ 这个包是 Java 2 1.5 通过 JSR166 引入的，是支持并发性等新功能的基础。它的作者 Doug Lea 是 JSR 的领导者。FJTask 框架不是 Java 2 1.5 的一部分，但在 [Lea00b] 可以使用它。

```
static void sort(final int[] A,final int lo, final int hi) {
   int n = hi - lo;
   if (n <= THRESHOLD){ Arrays.sort(A,lo,hi); return; }
   else {
      //split array
      final int pivot = (hi+lo)/2;

      //override run method in FJTask to execute run method
      FJTask t = new FJTask()
      { public void run()
         { sort(A, lo, pivot); }
      };

      //fork new task to sort lower half of array
      t.fork();

      //perform sort on upper half in current task
      sort(A,pivot,hi);

      //join with forked task
      t.join();

      //merge sorted arrays as before, code omitted
   }
 }
```

图 5-31 使用 FJTask 框架的合并排序程序

这个排序例程与前面的版本相似，但动态任务创建由 FJTask 子类的 run 方法（取代了 Thread 子类的 run 方法）实现。FJTask 的 fork 方法与 join 方法用于派生和聚合任务（替代了 Thread 的 start 和 join 方法）。尽管底层实现不同，但从程序员的角度来看，这种间接方法与前面所示的直接实现方法类似。

util.concurrent 类的 FJTask 示例中提供了合并排序的一种更先进的并行实现。这个包还包括在这个示例中没有演示的功能。

**知名应用**。FJTask 包的文档描述了几个使用派生 / 聚合模式的应用程序。最令人感兴趣的应用程序包括：Jacobi 迭代、并行分治矩阵乘法、标准并行处理基准程序（仿真了网格中的热扩散）、LU 矩阵分解、基于递归高斯积分法的积分计算和显微镜游戏变体[⊖]。

因为 OpenMP 基于派生 / 聚合编程模型，所以有人可能期望 OpenMP 程序员大量使用这种模式。但事实上，大多数程序员使用*循环并行*模式或者 SPMD 模式。这是因为当前的 OpenMP 标准并不能很好地支持并行区域的真正嵌套。关于标准 OpenMP 中使用这种模式的出版物很少，比较具有代表性的是一篇论述了在 LAPACK 的一种实现中使用嵌套并行来提供细粒度并行性的论文 [ARv03]。

对 OpenMP 进行扩展，使得它能够在大量应用程序中使用派生 / 聚合模式，已经成为一个活跃的研究领域。我们已经提到了一种为该模式提供间接映射解决方案的研究（任务队列 [SHPT00]）。另一种可能是利用显式线程组支持嵌套并行区域，为这种模式的直接映射提供解决方案（Nanos OpenMP）编译器 [GAM+00]。

170 ~ 172

⊖ 根据这个应用程序的文档，这是一个通过在《The 7th Guest》（T7G：用于 PC 的一个 CD — ROM 游戏）的实验室中的显微镜玩的游戏。它是一个棋盘游戏，两个玩家使用它们的卡片填充棋盘中的空格，有点类似翻转棋和奥赛罗游戏。

6. 相关模式

使用分治模式的算法使用了派生/聚合模式。

派生线程只处理单个并行循环的循环并行模式是派生/聚合模式的一个实例。

使用共享队列模式的主/从模式，可用于实现间接映射法。

## 5.8 共享数据模式

1. 问题

如何在一个并发任务集内显式地管理共享数据？

2. 背景

大多数算法结构模式通过一些技术，将共享数据“拉”到任务集之外，简单地处理共享数据。如任务并行模式中的复制和归约以及几何分解模式中的计算和通信的交替。但有些问题无法应用这些技术，因此需要在并发任务集内显式管理共享数据。

例如，考虑［YWC+96］中描述的分子生物学中的事物演化问题（phylogeny problem）。事物演化是一棵表示生物体之间关系的树。问题由所生成的作为潜在解决方案的大量子树（除去那些无法满足各种一致性条件的子树）组成。不同子树集可以并发检测，因此在并行事物演化算法中，任务自然定义为每一个子树集的处理过程。但是，并不需要对所有的子树集进行检测——如果集合 $S$ 被拒绝，则 $S$ 的所有超集也被拒绝。这样，就可以跟踪待检测的子树集和已经被拒绝的子树集。鉴于该算法很自然地分解为一些近乎独立的任务（每个集合对应一个），这个问题的解决方案可使用任务并行模式。但是，使用这种模式比较复杂，因为所有任务均需要读写被拒绝的子树集的数据结构。另外，因为这种数据结构在计算期间会发生改变，所以我们不能使用 5.6 节描述的复制技术。如使用几何分解模式，划分该数据结构，并基于这种数据分解设计一种解决方案，看上去好像是一种很好的选择。但是元素被拒绝的方式是不可预测的，因此任何类型的数据分解都有可能导致很差的负载均衡。

当必须在一个并发任务集中显式管理共享数据时，也会产生相似的难题。所有需要共享数据模式的问题具有以下共同点：程序执行过程中，至少一个数据结构被多个任务访问；至
173 少一个任务修改共享数据结构；在并发计算期间，任务可能需要使用修改值。

3. 面临的问题

- 面对计算期间可能产生的任意顺序的任务，计算结果必须正确。
- 显式地管理共享数据会产生并行开销，如果要高效运行程序，必须使这种开销尽可能小。
- 共享数据管理技术可能限制并发运行任务的数目，从而降低算法的可扩展性。
- 管理共享数据的结构如果无法很容易理解，程序的维护将非常困难。

4. 解决方案

在设计并行算法时，显式管理共享数据非常容易出错。因此，一种很好的方法是从强调简单性和抽象清晰性的解决方案开始。如果需要更优的性能，再尝试较复杂的解决方案。该解决方案反映这种方法。

**确保需要这种模式。**第一步确信真的需要这种模式。在设计过程中（例如，将问题分解为多个任务），回顾早期制定的策略，查看这些策略能否产生一种适合算法结构的某种模式的

解决方案，并且这种解决方案不需要显式管理共享数据，是非常有意义的。例如，如果任务并行模式非常适合，则值得分析设计，并查看是否可以通过复制和归约来处理依赖性。

**定义抽象数据类型**。假设必须使用这种模式，首先将共享数据定义为一种带有具体（可能很复杂的）操作集的抽象数据类型（ADT）。例如，如果共享数据结构是队列（参见 5.9 节），则操作集包括入队、出队、判断队列是否为空、判断一个指定元素是否存在等操作。每个任务通常会执行一系列操作，这些操作应当具有一个特点：如果它们串行执行（一次执行一个，不受其他任务的干扰），每个操作都必须保证数据的一致性。

每个操作的实现很可能包含一个底层操作序列，这些操作的结果对其他 UE 应当是不可见的。例如，如果使用链表实现前面提到的队列，则出队操作实际上包含一个较低层操作序列（而较低层操作本身由一个更低层操作序列组成）。

1）根据变量 `first` 获得列表中第一个对象的引用。

2）根据第一个对象，获得列表中第二个对象的引用。

3）将 `first` 值修订为第二个对象的引用。 174

4）更新列表的大小。

5）返回第一个元素。

如果两个任务并发执行出队操作，而且这些较低层的操作交错执行（出队操作不是以原子方式执行），则很容易产生不一致的链表。

**实现一种合理的并发控制协议**。在确定了 ADT 及其操作集之后，还要实现一种并发控制协议，以确保这些操作能像串行地执行一样给出相同结果。这可以通过多种方式实现，首先使用一种最简单的技术，如果该技术无法满足性能需求，再尝试使用一些更复杂的技术。若一种较复杂的技术依然无法满足需求，可以组合使用这些技术。

**一次执行一个操作**。最简单的解决方案是确保所有操作串行执行。

在共享内存环境中，实现这种方法最简单的方式是将每一个操作都作为临界区的一部分，并使用互斥协议确保一次只有一个 UE 正在执行。这意味着所有对数据的操作都是互斥的。如何实现这种方式依赖于目标编程环境的设施。典型的方法包括互斥锁、同步块、临界区和信号量。这些机制在第 6 章中有详细描述。如果编程语言本身支持这些抽象数据类型的实现，一个合理的做法是将每一个操作实现为一个过程或方法，并且在方法本身中实现互斥协议。

在一个消息传递环境中，确保串行执行的最简单方式是将共享数据结构分配给一个特定 UE。每一个操作对应于一种消息类型，其他进程通过向管理该数据结构的 UE 发送消息申请操作，管理数据结构的 UE 串行处理这些请求。

在这两种环境中，实现这种方式并不困难。但这种方式太过保守（甚至不允许并发执发能够安全同步执行的操作），并且它可能会成为性能瓶颈。如果发生这种情况，应当对本节所描述的其他方法进行评估，找到一种能够降低或消除这种瓶颈并提供更优性能的方法。

**非干扰的操作集合**。分析操作间的干扰性是提高性能的一种方法。如果操作 A 对操作 B 将要读取的一个变量进行写操作，则认为操作 A 干扰操作 B。如果多个任务执行相同操作（例如，多个任务对一个共享队列执行出队操作），则认为操作存在自干扰。所有操作可能划分为两个不相交的集合，不同集合的操作互不干扰。在这种情况下，可将每一个集合定义为 175
一个临界区，以增加并发性。即在集合内部一次执行一个操作，但不同集合的操作可以并发执行。

**读访问者 / 写访问者**。如果没有一种简单方法将操作划分为不相交的集合，则考虑干扰类型。可能会出现这样一种情况，某些操作修改数据，而其他操作仅仅读取数据。例如，如果操作 A 是一个写访问者（既读数据又写数据），操作 B 是一个读访问者（只读数据），操作 A 干扰自己和操作 B，但操作 B 不干扰自己。如果一个任务正在执行操作 A，则不应该有其他任务正在执行操作 A 或操作 B。但任意数量的任务都应当能并发执行操作 B。在这种情况下，值得实现一种读访问者 / 写访问者协议，该协议将允许挖掘这种潜在的并发性。注意，管理读访问者 / 写访问者协议的开销大于管理简单互斥锁的开销，因此对读访问者访问数据的计算应当足够复杂，使得这种开销具有价值。另外，读访问者的数目应该远大于写访问者。

`java.util.concurrent` 包提供了支持读访问者 / 写访问者协议的读 / 写锁。图 5-32 中的代码演示了这些锁的一般使用方式：首先实例化 `ReadWriteLock`，然后获得其读写锁。`ReentrantReadWriteLock` 是一个实现 `ReadWriteLock` 接口的类。当执行读操作时，将读锁上锁。当执行写操作时，将写锁上锁。锁的语义是：任意数目的 UE 可以同时拥有读锁，但写锁是专有的。即仅有一个 UE 可以拥有写锁，并且如果写锁被某一 UE 拥有，其他 UE 也就不能拥有读锁。

```
class X {
  ReadWriteLock rw = new ReentrantReadWriteLock();
  // ...

  /*operation A is a writer*/
  public void A() throws InterruptedException {
    rw.writeLock().lock(); //lock the write lock
    try {
      // ... do operation A
    }
    finally {
      rw.writeLock().unlock(); //unlock the write lock
    }
  }

 /*operation B is a reader*/
 public void B() throws InterruptedException {
    rw.readLock().lock(); //lock the read lock
    try {
      // ... do operation B
    }
    finally {
      rw.readLock().unlock(); //unlock the read lock
    }
  }
}
```

图 5-32 读 / 写锁的典型使用方式。这些锁定义在 `java.util.concurrentlocks` 包中。在 `finally` 块中调用 `unlock` 方法，以确保无论 `try` 块如何退出（正常或异常退出），该锁都将被解锁。这是 Java 程序使用锁（而不是同步块）的一种标准惯例

在 [And00] 和大多数操作系统书籍中都讨论了读访问者 / 写访问者协议。

**缩减临界区的大小**。更详细地分析操作的实现是提高性能的另一种方法。可能会出现这样一种情况：操作仅有某一部分的子操作会与其他操作相干扰。如果这样，则可缩减临界区的大小。注意，这种优化非常容易出错。因此，仅当这种方法能够显著提高性能并且程序员对操作间的干扰完全理解时，才能应尝试使用。

**嵌套锁**。这种方法是前面两种方法（非干扰操作和缩减临界区大小）的一种混合技术。假设定义一种具有两种操作的 ADT。操作 A 首先多次读并更新变量 x，然后在单条语句中读并更新变量 y。操作 B 读并写 Y。有些分析认为：所有执行 A 的 UE 需要互斥，所有执行 B 的 UE 也需要互斥。又因为这两种操作都读并更新变量 y，所以操作 A 和 B 也需要互斥。但通过进一步分析会发现，这两种操作几乎是非干扰的。如果不是因为一条语句（在该语句中，A 读并更新 y），这两种操作可以在各自的临界区中实现，这将使 A 和 B 能够并发执行。图 5-33 展示了一种使用两个锁的解决方案：A 为整个操作获取并拥有 lockA，B 为整个操作获取并拥有 lockB，A 仅为更新 Y 的语句获取并拥有 lockB。

```
class Y {
  Object lockA = new Object();
  Object lockB = new Object();

  void A()
  { synchronized(lockA)
    {
      ....compute....
      synchronized(lockB)
      { ....read and update y....
      }
    }
  }

  void B() throws InterruptedException
  { synchronized(lockB)
    { ...compute....
    }
  }
}
```

图 5-33 嵌套锁示例，synchronized 代码块定义了两个虚拟对象：lockA 和 lockB

当使用嵌套锁时，程序员应仔细复查代码以防止死锁。上面例子中死锁的典型示例是：首先 A 获得 lockA，B 获得 lockB；然后 A 尝试获取 lockB，B 尝试获取 lockA。现在两个操作都无法继续执行。解决死锁的方法为：定义一种锁偏序，并确保操作总是以偏序所指定的顺序获取锁。在前面的示例中，假设我们定义锁偏序 lockA<lockB，并确保 lockA 永远不会被一个已经获取 lockB 的 UE 所获取。

**特定应用的语义释放**。另外一种方法是复制部分共享数据（[YWC+96] 中描述的软件缓存），在不影响计算结果的前提下，甚至允许这些副本不一致。例如，前面描述的事物演化问题的分布式存顺储器解决方案，可以将被拒绝的集合副本赋给每个 UE，并且允许这些副本不同步；任务可以做其他额外的工作（拒绝一个已经被其他 UE 上的一个任务所拒绝的集合），但这些额外的工作不会影响计算结果，并且与保持所有副本同步的通信开销相比，它更高效。

**回顾其他考虑事项**。内存同步。确保内存同步（根据需要）：缓存和编译器优化会导致与共享变量相关的不期望的行为。例如，从缓存或寄存器中读到的是变量的旧值，而不是另一个任务写入的最新值，或者最新值还没有写入内存中（在这种情况下，该值对其他任务不可见）。虽然大多数情况下，存储器同步由较高级别的同步原语隐式完成，但仍需关注这个问题。遗憾的是，存储器同步技术与平台特别相关。例如，在 OpenMP 中，可以使用 flush 指令显式地同步内存（该指令将被其他几条指令隐式调用）；在 Java 中，当进入和离开一个同步

区域时，存储器被隐式同步；在 Java 2 1.5 环境下，当执行加锁和解锁操作时，存储器被隐式同步。另外，将变量标识为 volatile，相关存储器也被隐式同步。第 6 章将更详细地讨论这一点。

**任务调度**。这种模式对数据依赖性的显式管理是否会影响任务调度？实现良好的负载均衡是任务调度的主要目标。除了在第 4 章中所描述的因素之外，还应当考虑任务可能会因为等待访问共享数据而被悬挂。可以尝试一种能够最小化这种等待的任务分配方式，或者为每个 UE 分配多个任务，期望每一个 UE 上总是存在一个不等待访问共享数据的任务。

5. 示例

**共享队列**。共享队列是一种经常使用的 ADT，而且是*共享数据*模式的一个非常好的示例。5.9 节讨论了并发控制协议和高效共享队列的技术。

**用于非线性优化的遗传算法**。考虑 SPEC OMP2001 基础套件 [ADE+01] 中的 GAFORT 程序。GAFORT 是一个小 Fortran 程序（大约 1500 行），实现了一个用于非线性优化的遗传算法。该算法的计算主要由整数运算组成，程序性能瓶颈是需要在存储器子系统中移动大量数据数组。

对本节内容来说，遗传算法的细节并不重要。基于 [EM] 中对 GAFORT 的讨论，我们将重点考虑 GAFORT 中的某个循环，其串行版本的伪代码如图 5-34 所示。该循环对所有染色体重新排序，其运行时间占到一个典型 GAFORT 作业的大约 36% [AR031]。

```
Int const NPOP // number of chromosomes (~40000)
Int const NCHROME // length of each chromosome

Real :: tempScalar
Array of Real :: temp(NCHROME)
Array of Int :: iparent(NCHROME, NPOP)
Array of Int :: fitness(NPOP)
Int :: j, iother

loop [j] over NPOP
    iother = rand(j) // returns random value greater
                     // than or equal to zero but not
                     // equal to j and less than NPOP

    // Swap Chromosomes
    temp(1:NCHROME) = iparent(1:NCHROME, iother)
    iparent(1:NCHROME, iother) = iparent(1:NCHROME, j)
    iparent(1:NCHROME, j) = temp(1:NCHROME)

    // Swap fitness metrics
    tempScalar = fitness(iother)
    fitness(iother) = fitness(j)
    fitness(j) = tempScalar

end loop [j]
```

图 5-34 遗传算法程序 GAFORT 中对所有染色体进行重新排序的伪代码

该程序的一个并行版本将使用*循环并行*模式来并行化该循环。在这个示例中，`iparent` 和 `fitness` 数组为共享数据。循环体内包含这两个数组的计算由交换 `iparent` 数组的两个元素和交换 `fitness` 的对应元素组成。对这些操作的分析表明，当在两个交换操作中至少有一个相同的交换位置时，这两个操作为“干扰”操作。

将共享数据看作一个 ADT，将有助于识别并分析对共享数据的操作。但是，这并不意味着实现本身始终需要反映这种结构。在某些情形中，特别是数据结构比较简单且编程语言不能很好地支持 ADT 时，放弃 ADT 所隐含的封装并直接处理数据，可能更高效。本示例就演示了这种情形。

前面提到，交换染色体操作可能是彼此“干扰”的：因此作用在 j 上的循环不能安全并行执行。最简单的方法是使用临界区强制执行“一次只执行一个操作”协议，如图 5-35 所示。也需要修改随机数产生器，使得当它被多个线程并行调用时，能够产生一致的伪随机数集合。完成这个任务的算法很好理解 [Mas971]，这里不再讨论。

```
#include <omp.h>
   Int const NPOP // number of chromosomes (~40000)
   Int const NCHROME // length of each chromosome

   Real :: tempScalar
   Array of Real :: temp(NCHROME)
   Array of Int :: iparent(NCHROME, NPOP)
   Array of Int :: fitness(NPOP)
   Int :: j, iother

#pragma omp parallel for
   loop [j] over NPOP
      iother = par_rand(j) // returns random value greater
                           // than or equal to zero but not
                           // equal to j and less than NPOP

#pragma omp critical
{
      // Swap Chromosomes
      temp(1:NCHROME) = iparent(1:NCHROME, iother)
      iparent(1:NCHROME, iother) = iparent(1:NCHROME, j)
      iparent(1:NCHROME, j) = temp(1:NCHROME)

      // Swap fitness metrics
      tempScalar = fitness(iother)
      fitness(iother) = fitness(j)
      fitness(j) = tempScalar
}
   end loop [j]
```

图 5-35　遗传算法程序 GAFORT 中对所有染色体进行并行重新排序算法的低效方法的伪代码

图 5-35 中的代码可以通过多个线程运行，但当线程数目不断增多时，程序性能将不会提高。事实上，线程数目过多时，程序性能会降低。这是因为当线程在临界区等待执行时，它们会浪费系统资源。事实上，并发控制协议消除了所有的可用并发性。

该问题的解决方案基于以下事实：仅当两个对共享数据的交换操作中至少有一个相同的交换位置时，这两个操作才是“干扰”的。因此，使用嵌套锁并发控制协议，当循环迭代操作不“干扰”时，仅会添加少量的开销。[ADE+01] 中使用的方法是为每一个染色体创建一个 OpenMP 锁。图 5-36 展示了这种方法的伪代码。在最终程序中，实际上大多数循环迭代不相互“干扰”。染色体总数 NPOP（在 SPEC OMP2001 基准测试中是 40 000 个）远大于 UE 的数目，循环迭代间相互干扰的几率非常小。

```
#include <omp.h>
    Int const NPOP // number of chromosomes (~40000)
    Int const NCHROME // length of each chromosome

    Array of omp_lock_t :: lck(NPOP)

    Real :: tempScalar
    Array of Real :: temp(NCHROME)
    Array of Int :: iparent(NCHROME, NPOP)
    Array of Int :: fitness(NPOP)
    Int :: j, iother

// Initialize the locks
#pragma omp parallel for
    for (j=0; j<NPOP; j++){ omp_init_lock (&lck(j)) }

#pragma omp parallel for
    for (j=0; j<NPOP; j++){
        iother = par_rand(j) // returns random value >= 0, != j,
                             // < NPOP
        if (j < iother) {
           set_omp_lock (lck(j)); set_omp_lock (lck(iother))
        }
        else {
           set_omp_lock (lck(iother)); set_omp_lock (lck(j))
        }

        // Swap Chromosomes
        temp(1:NCHROME) = iparent(1:NCHROME, iother);
        iparent(1:NCHROME, iother) = iparent(1:NCHROME, j);
        iparent(1:NCHROME, j) = temp(1:NCHROME);

        // Swap fitness metrics
        tempScalar = fitness(iother)
        fitness(iother) = fitness(j)
        fitness(j) = tempScalar

        if (j < iother) {
           unset_omp_lock (lck(iother)); unset_omp_lock (lck(j))
        }
        else {
           unset_omp_lock (lck(j)); unset_omp_lock (lck(iother))
        }
    } // end loop [j]
```

图 5-36 遗传算法程序 GAFORT 中对所有染色体进行重新排序算法的并行循环的伪代码。这个版本为每个染色体都使用一个独立的锁，并能高效地并行执行

OpenMP 锁的详细描述见附录 A。OpenMP 锁为一种不透明的数据类型：`omp_lock _t`，该类型定义在 `omp.h` 头文件中。在上面的示例中，锁数组定义在一个独立的并行循环中，并被初始化。在染色体交换循环中，首先为每对被交换的染色体设置锁，然后执行交换操作，最后解锁。因为使用了嵌套锁，所以必须考虑出现死锁的可能性。这里的解决方案是使用与铠关联的数组元素索引值来对锁排序。当一对循环迭代碰巧在同一时间交换两个相同元素时，按照这种顺序获取锁能够阻止死锁的发生。在更高效的并发控制协议实现后，程序能很好地并行运行。

**知名应用**。[YWC+96] 中描述了背景部分中所讨论的事物演变问题的一种解决方案。总体方法适合任务并行模式；使用复制和周期更新方法来显式管理已被拒绝集合的数据结构，周期更新是为了重新建立副本间的一致性。

[YWC+96] 中介绍的另外一个问题是 Gröbner 基本程序。在这个应用程序中，省略了大部

分细节，计算由下面几个部分组成：首先使用多项式对产生新的多项式，然后将它们与一个多项式主集合进行比较，最后将那些不是由主集合中的元素线性组合而成的多项式添加到主集合中（主集合中的元素用于生成新的多项式对）。由于不同的多项式对可以并发处理，因此可以为每一个多项式对定义一个任务，并将它们分配给 UE。[YWC+96] 中描述的解决方案适合*任务并行模式*（具有一个由多项式对组成的任务队列），并且使用一种称为*软件缓存*的特定应用程序协议来显式地管理主集合。

6. 相关模式

5.9 节和 5.10 节讨论特定的共享数据结构类型。许多使用*共享数据*模式的问题为它们的算法结构使用了*任务并行*模式。

176 ≀ 182

## 5.9 共享队列模式

1. 问题

并行执行的 UE 如何安全共享队列数据结构？

2. 背景

许多并行算法的高效实现需要构建将被所有 UE 共享的队列数据结构。最常见的情况是使用*主 / 从*模式的程序。

3. 面临的问题

- 简单的并发控制协议提供了较好的抽象清晰性，这使得程序员更容易验证共享队列实现的正确性。
- 如果在单个同步结构中包含过多的共享队列，并发控制协议将增加 UE 被阻塞的几率（UE 等待访问共享队列），并减少并发性。
- 如果要实现并发控制协议与共享队列很好的协调，并增加并发性，需要更复杂但更易出错的同步结构。
- 具有复杂内存层次的计算机系统（如 NUMA 机器和集群）维持单个队列会导致大量通信，并增加并行开销。因此，可能需要划分队列抽象，使用多个或分布式队列。

4. 解决方案

理想情况下，共享队列应作为目标编程环境的一部分，要么显式实现为程序员可用的一种 ADT，要么隐式实现为支持使用共享队列的较高级别模式（如*主 / 从*模式）。在 Java 2 1.5 中，可在 `java.util.concurrent` 包中获得这样的队列。本节内容将从零开始开发实现，以演示这些概念。

共享队列的实现可能非常棘手。首先必须应用合理的同步以避免竞争条件；其次还有性能考虑（特别是当大量 UE 访问共享队列时）；再次，可能需要更复杂的同步操作；最后，在某些情况下，可能需要分布式队列，以消除性能瓶颈。

然而，如果需要实现共享队列，可以将其实现为*共享数据*模式的一个实例：首先定义队列的 ADT，包括队列值及其操作集合；然后定义并发控制协议，从最简单的“一次执行一个操作”的解决方案开始，并逐步改善。为使讨论更为具体，我们将从一个特定问题出发定义队列：在一个*主 / 从*模式算法中实现一个存储任务的队列。尽管如此，这里讨论的解决方案是通用的，可以方便地进行扩展，以应用于共享队列的其他应用中。

183

**抽象数据类型（ADT）**。ADT 包含一个值集以及定义在这些值集上的操作。对于队列来

说，值集是包含某种类型的0个或多个对象的有序列表（例如，整数或者任务ID）。队列有入队（put）和出队（take）两个操作。在某些情况下，队列可能会包含更多的操作。但对于本节的讨论来说，这两种操作已经足够了。

我们必须明确当对一个空队列执行出队操作时的处理方式。这依赖于主/从模式算法所采用的终止方式。例如，假设所有任务都是master启动时创建的，那么空任务队列意味着该UE应当终止操作。我们希望空队列的出队操作能够根据一个标记队列为空的标识立即返回，即非阻塞队列。另外一种可能的情况是，任务是动态创建的，只有接收到一个特定poison-pill任务时，UE才终止操作。在这种情况下，对于空队列的出队操作来说，合理的行为应该是等待直到队列非空。即当队列为空时，阻塞队列访问。

**“一次只执行一个操作”的队列**

**非阻塞队列**。因为队列将被并发访问，所以必须定义并发控制协议，以确保不会发生多个UE相互干扰的情况。如5.8节所述，最简单的解决方案是使所有ADT操作互斥。因为队列操作不会被阻塞，所以可以采用互质的一种简单实现方式（在第6章中有详细描述）。图5-37所示的Java实现使用链表来存放队列中的任务（我们开发了自己的列表类，以演示如何添加合理的同步操作。所以并没有使用其他已有的非同步类库，如`java.util.LinkedList`或者`java.util.concurrent`包中一个类）。其中`head`指向一个始终存在的虚拟节点[⊖]。

```
public class SharedQueue1
{
   class Node //inner class defines list nodes
   { Object task;
      Node next;

      Node(Object task)
      {this.task = task; next = null;}
   }

   private Node head = new Node(null); //dummy node
   private Node last = head;

   public synchronized void put(Object task)
   { assert task != null: "Cannot insert null task";
      Node p = new Node(task);
      last.next = p;
      last = p;
   }

   public synchronized Object take()
   { //returns first task in queue or null if queue is empty
      Object task = null;
      if (!isEmpty())
      { Node first = head.next;
         task = first.task;
         first.task = null;
         head = first;
      }
      return task;
   }

   private boolean isEmpty(){return head.next == null;}
}
```

图5-37 该队列确保一次最多有一个线程能够访问。如果队列为空，立即返回`null`

⊖ `take`操作使原head节点成为虚拟节点，而不是简单的操作`next`指针。这使得后面可以优化代码，使`put`和`take`操作并发执行。

队列中的第一个任务（如果存在）存放在 head.next 所指向的节点中。isEmpty 函数是私有的，仅在同步函数的内部调用。这样实现的好处是它不需要同步操作（如果是公共的，则它也需要同步操作）。当然，任务结构的实现存在很多方式。

**当队列为空时，阻塞访问。**图 5-38 展示了共享队列的第二个版本。该版本修改了 take 操作：当线程尝试对空队列执行 take 操作时，不是立即返回而是等待，直到新任务出现。线程等待时释放它所拥有的锁，在尝试进行下一次操作之前，再重新获取它。Java 使用 wait 和 notify 方法完成这个功能，相关详细描述见附录 C。附录 C 还展示了使用 java.util.concurrent.locks 包中的锁实现的队列。该包由 Java 2 1.5 引入，用于替代 wait 和 notify 函数。POSIX 线程（Pthread）[But97、IEE] 可以使用相似原语，读者可以在 [And00] 中找到利用信号量实现这种功能的技术和其他的一些基本原语。

```
public class SharedQueue2
{
   class Node
   { Object task;
     Node next;

     Node(Object task)
     {this.task = task; next = null;}
   }

   private Node head = new Node(null);
   private Node last = head;

   public synchronized void put(Object task)
   { assert task != null: "Cannot insert null task";
     Node p = new Node(task);
     last.next = p;
     last = p;
     notifyAll();
   }

   public synchronized Object take()
   { //returns first task in queue, waits if queue is empty
     Object task = null;
     while (isEmpty())
       {try{wait();}catch(InterruptedException ignore){}}
     { Node first = head.next;
       task = first.task;
       first.task = null;
       head = first;
     }
     return task;
   }

   private boolean isEmpty(){return head.next == null;}
}
```

图 5-38 共享队列确保每次只有一个线程可以访问数据结构。与第一个共享队列示例不同，如果队列为空，线程将等待。当该共享队列用于主 / 从模式算法时，将需要定义 poison pill 任务，用于终止线程操作

一般来说，为了将一个“如果条件为假就立即返回”的方法改变为一个“等待直到条件为真”的方法，需要做两处改变：首先，将下面形式的语句

```
if (condition){do_something;}
```

替换为一个循环[⊖]：

```
while( !condition){wait();} do_something;
```

其次，检查共享队列的其他操作，在所有可能设立条件的操作中添加对 notifyAll 函数的调用。这是 wait 函数基本用法的一个实例。附录 C 更详细地描述这一点。

这样，图 5-38 中的代码主要做了两处改动。首先，将代码

```
if (!isEmpty()){....}
```

替换为

```
while(isEmpty())
  {try{wait()}catch(InterruptedException ignore){}}{....}
```

其次，注意，put 函数将使队列不为空，因此在该方法中添加对 notifyAll 函数的调用。

这种实现存在性能问题，即存在对 notifyAll 函数的额外调用。虽然不影响正确性，但它可能降低性能。一种优化方法是最小化 put 函数中对 notifyAll 函数的调用数量。实现这种功能的一种方式是跟踪等待线程的数目，仅当存在等待线程时，才进行一次 notifyAll 函数的调用。我们使用整数 w 标记等待线程数量，如下所示：

```
while( !condition){w++; wait(); w--} do_something;
```

和

```
if (w>0) notifyAll();
```

在这个特定示例中，因为仅有一个等待线程将能使用任务，所以可以使用 notify 函数替换 notifyAll 函数（notify 函数仅通知一个等待线程）。我们将在后面的示例（见图 5-40）中演示这种细化的代码。

184 ~ 187

**用于非干扰操作的并发控制协议**。如果共享队列的性能不能满足要求，就必须定义更高效的并发控制协议。像 5.8 节所讨论的那样，需要定义 ADT 的非干扰操作集合。仔细分析非阻塞共享队列的操作（参见图 5-37 和图 5-38），会发现 put 和 take 操作总是访问不同变量，因此这两个操作是非干扰的。put 操作修改了 last 引用和 last 所指向对象的 next 成员。take 操作修改了 head 引用和 head.next 所指向对象的 task 成员的值。在这种情况下，put 修改了 last 和某个 Node 对象的 next 成员。而 take 修改了 head 和某个对象的 task 成员。这些操作都是非干扰的，因此可以为 put 和 next 操作分别定义不同的锁。解决方案如图 5-39 所示。

**使用嵌套锁的并发控制协议**。图 5-39 所示的方法无法方便地应用到一个“当队列为空时阻塞访问”的队列中。首先，wait、notify 和 notifyAll 函数仅能够在队列的一个同步块中调用。其次，如果优化 notify 调用（如前所述），则 w（即等待线程的数目）将被 put 和 take 操作访问。因此，需要定义锁（putLock）来保护 w，并在队列为空时阻塞执行 take 操作的线程。代码如图 5-40 所示。注意，get 中的 putLock.wait() 仅释锁 putLock。因此一个被阻塞线程的其他 take 操作（来自使用 takeLock 的外层同步区域）

⊖ 事实上，wait 方法将抛出 InterruptedException 异常，该异常必须处理。这里为简单起见，并没有对此进行处理。然而，在示例代码中对其进行了适当处理。

将会被继续阻塞。对于这个特定问题来说，这是可行的。这个策略在队列不为空时允许 put 和 take 操作并发执行。

```
public class SharedQueue3
{
   class Node
   { Object task;
      Node next;

      Node(Object task)
      {this.task = task; next = null;}
   }

   private Node head = new Node(null);
   private Node last = head;

   private Object putLock = new Object();
   private Object takeLock = new Object();

   public void put(Object task)
   { synchronized(putLock)
      { assert task != null: "Cannot insert null task";
         Node p = new Node(task);
         last.next = p;
         last = p;
      }
   }

   public Object take()
   { Object task = null;
      synchronized(takeLock)
      { if (!isEmpty())
         { Node first = head.next;
            task = first.task;
            first.task = null;
            head = first;
         }
      }
      return task;
   }
}
```

图 5-39 实现了 put 和 take 非干扰操作的共享队列，并通过定义独立的锁，使它们能并发执行

```
pubic class SharedQueue4
{
   class Node
   { Object task;
      Node next;

      Node(Object task)
      {this.task = task; next = null;}
   }

   private Node head = new Node(null);
   private Node last = head;
   private int w;
   private Object putLock = new Object();
   private Object takeLock = new Object();

   public void put(Object task)
   { synchronized(putLock)
      { assert task != null: "Cannot insert null task";
```

图 5-40 使用多个锁的阻塞队列，当队列非空时，put 和 take 操作能够并发执行

```
        Node p = new Node(task);
        last.next = p;
        last = p;
        if(w>0){putLock.notify();}
      }
    }
    public Object take()
    { Object task = null;
      synchronized(takeLock)
      { //returns first task in queue, waits if queue is empty
        while (isEmpty())
       { try{synchronized(putLock){w++; putLock.wait();w--;} }
          catch(InterruptedException error){assert false;}}
        { Node first = head.next;
          task = first.task;
          first.task = null;
          head = first;
        }
      }
      return task;
    }

    private boolean isEmpty(){return head.next == null;}

  }
```

图 5-40 （续）

另外一个需要注意的问题是，该解决方案在 take 和 put 操作中均使用了同步块。因此，应当仔细检查同步块以防止潜在的死锁。然而，在这个示例中，死锁并不存在。这是因为 put 操作仅使用一个锁：putLock。但为了更通用，本示例定义了一个锁偏序，并确保线程始终以该偏序所定义的顺序获取锁。例如，假设定义锁偏序：takeLock<putLock，并确保线程以该偏序所定义的顺序进入同步块。

如前所述，基于 Java 的实现的几种队列包含在 Java2 1.5 中的 java.util.concurrent 包中。其中一些队列的实现基于这里讨论的最简单策略，而另一些队列的实现基于更复杂的策略，这些复杂策略提供了额外的灵活性和性能。

**分布式共享队列**。集中式共享队列可能会成为性能瓶颈，定义并实现分布式共享队列可突破这个瓶颈。作为一个示例，我们将在底层实现中开发一个支持派生 / 聚合程序（使用线程池）的简单代码包和一个分布式任务队列。该包是 FJTask 包 [Lea00b]（基于 [BJK+96] 中的思想）的简化版本。其基本思想是创建一个线程池（执行程序运行时动态创建的任务），每一个线程都关联一个非阻塞队列（替代单个集中式任务队列），并将新产生的任务放在自己的队列中。当线程需要执行新任务时，它首先尝试从自己的队列中获取任务。如果队列为空，该线程随机选择另外一个线程，并尝试从那个线程的队列中偷取任务。如果任务获取失败，该线程将继续检测其他队列，直到找到一个新任务为止（在 [BJK+96] 中，这称为随机工作偷取）。

当线程接收到 poison-pill 任务时，它将终止操作运行。对于派生 / 聚合程序，需要牢记的是：当线程以 LIFO（后进先出）顺序从自己的队列中移除任务，并以 FIFO（先进先出）顺序从其他队列中移除任务时，已经证明该方法能够很好地工作。因此，我们将为 ADT 添加一个新操作：移除队列的最后一个元素。线程可调用该操作从自己的队列中移除任务。如图 5-41 所示，该实现与图 5-40 相似，但具有一个额外的操作：takeLast。

```
public class SharedQueue5
{
   class Node
   { Object task;
      Node next;
      Node prev;

      Node(Object task, Node prev)
      {this.task = task; next = null; this.prev = prev;}
   }

   private Node head = new Node(null, null);
   private Node last = head;

   public synchronized void put(Object task)
   { assert task != null: "Cannot insert null task";
      Node p = new Node(task, last);
      last.next = p;
      last = p;
   }

   public synchronized Object take()
   { //returns first task in queue or null if queue is empty
      Object task = null;
      if (!isEmpty())
      { Node first = head.next;
         task = first.task;
         first.task = null;
         head = first;
      }
      return task;
   }

   public synchronized Object takeLast()
   { //returns last task in queue or null if queue is empty
      Object task = null;
      if (!isEmpty())
      { task = last.task; last = last.prev; last.next = null;}
      return task;
   }

   private boolean isEmpty(){return head.next == null;}
}
```

图 5-41 包含 takeLast 操作的非阻塞共享队列

代码包的其余部分由 3 个类组成。

- Task 类。一个抽象类。应用可扩展它并重写 run 方法，以定义任务功能。这个类提供 fork 和 join 等方法。
- TaskRunner 类。该类扩展 Thread 类，添加线程池中线程的功能。该类的每个实例包含一个共享任务队列，任务偷取代码就位于这个类中。
- TaskRunnerGroup 类。该类管理 TaskRunner 对象，并定义初始化和关闭线程池的方法以及 executeAndWait 方法（启动一个任务的运行，并等待该任务的完成）。executeAndWait 方法用于启动计算（之所以需要它，是因为 Task 类中的 fork 方法仅能够在一个 Task 对象内被调用。我们将在后面描述这种限制的原因）。

现在进一步讨论这些类。图 5-42 展示了 Task 类。与该抽象类相关的状态是标记为 volatile 的变量 done，以保证访问它的任何线程都将获得最新值。

```
public abstract class Task implements Runnable
{
  //done indicates whether the task is finished
   private volatile boolean done;
   public final void setDone(){done = true;}
   public boolean isDone(){return done;}

  //returns the currently executing TaskRunner thread
   public static TaskRunner getTaskRunner()
   { return (TaskRunner)Thread.currentThread(); }

  //push this task on the local queue of current thread
   public void fork()
   { getTaskRunner().put(this);
   }

  //wait until this task is done
   public void join()
   { getTaskRunner().taskJoin(this);
   }

  //execute the run method of this task
   public void invoke()
   { if (!isDone()){run(); setDone(); }
   }
}
```

图 5-42 Task 抽象基类

图 5-43、图 5-44 和图 5-45 展示了 TaskRunner 类。如在方法 run 中所定义的那样，线程将不断循环，直到遇到 poison 任务。线程首先尝试从自己的局部队列的后部获得任务。如果该局部队列为空，则尝试从其他线程队列的前面偷取一个任务。

```
import java.util.*;

class TaskRunner extends Thread
{

   private final TaskRunnerGroup g; //managing group
   private final Random chooseToStealFrom; //random number generator
   private final Task poison; //poison task
   protected volatile boolean active; //state of thread
   final int id; //index of task in the TaskRunnerGroup

   private final SharedQueue5 q; //Nonblocking shared queue

   //operations relayed to queue
   public void put(Task t){q.put(t);}
   public Task take(){return (Task)q.take();}
   public Task takeLast(){return (Task)q.takeLast();}

   //constructor
   TaskRunner(TaskRunnerGroup g, int id, Task poison)
   { this.g = g;
      this.id = id;
      this.poison = poison;
      chooseToStealFrom = new Random(System.identityHashCode(this));
      setDaemon(true);
      q = new SharedQueue5();
   }

   protected final TaskRunnerGroup getTaskRunnerGroup(){return g;}
   protected final int getID(){return id;}
/* continued in next figure */
```

图 5-43 TaskRunner 类，该类定义了线程池中线程的行为（续见图 5-44 和图 5-45）

```
/* continued from previous figure */
   //Attempts to steal a task from another thread. First chooses a
   //random victim, then continues with other threads until either
   //a task has been found or all have been checked. If a task
   //is found, it is invoked. The parameter waitingFor is a task
   //on which this thread is waiting for a join. If steal is not
   //called as part of a join, use waitingFor = null.
   void steal(final Task waitingFor)
   { Task task = null;

      TaskRunner[] runners = g.getRunners();
      int victim = chooseToStealFrom.nextInt(runners.length);
          for (int i = 0; i != runners.length; ++i)
        { TaskRunner tr = runners[victim];
          if (waitingFor != null && waitingFor.isDone()){break;}
        else
        { if (tr != null && tr != this)
          task = (Task)tr.q.take();
          if(task != null) {break;}
          yield();
          victim = (victim + 1)%runners.length;

        }
       } //have either found a task or have checked all other queues

       //if have a task, invoke it
       if(task != null && ! task.isDone())
       { task.invoke(); }
   }
/* continued in next figure
```

图 5-44 TaskRunner类，该类定义了线程池中线程的行为（续图 5-43 和续见图 5-45）

```
/* continued from previous figure */
  //Main loop of thread. First attempts to find a task on local
  //queue and execute it. If not found, then tries to steal a task
  //from another thread. Performance may be improved by modifying
  //this method to back off using sleep or lowered priorities if the
  //thread repeatedly iterates without finding a task. The run
  //method, and thus the thread, terminates when it retrieves the
  //poison task from the task queue.
   public void run()
   { Task task = null;
      try
      { while (!poison.equals(task))
         { task = (Task)q.takeLast();
           if (task != null) { if (!task.isDone()){task.invoke();}}
           else { steal(null); }
         }
       } finally { active = false; }
    }

   //Looks for another task to run and continues when Task w is done.
    protected final void taskJoin(final Task w)
    { while(!w.isDone())
       { Task task = (Task)q.takeLast();
          if (task != null) { if (!task.isDone()){ task.invoke();}}
          else { steal(w);}
       }
    }
}
```

图 5-45 TaskRunner类，该类定义了线程池中线程的行为（续图 5-43 和图 5-44）

TaskRunnerGroup类的代码如图5-46所示。TaskRunnerGroup类的构造方法以线程数目作为参数，初始化线程池。通常情况下，应根据计算机系统中处理器的数目合理选择该值。executeAndWait方法启动一个任务并把它放到线程0的任务队列中。

```
class TaskRunnerGroup
{ protected final TaskRunner[] threads;
  protected final int groupSize;
  protected final Task poison;

  public TaskRunnerGroup(int groupSize)
  { this.groupSize = groupSize;
    threads = new TaskRunner[groupSize];
    poison = new Task(){public void run(){assert false;}};
    poison.setDone();
    for (int i = 0; i!= groupSize; i++)
      {threads[i] = new TaskRunner(this,i,poison);}
    for(int i=0; i!= groupSize; i++){ threads[i].start(); }
  }

  //start executing task t and wait for its completion.
  //The wrapper task is used in order to start t from within
  //a Task (thus allowing fork and join to be used)
  public void executeAndWait(final Task t)
  { final TaskRunnerGroup thisGroup = this;
    Task wrapper = new Task()
      { public void run()
        { t.fork();
          t.join();
          setDone();
          synchronized(thisGroup)
            { thisGroup.notifyAll();} //notify waiting thread
        }
      };
    //add wrapped task to queue of thread[0]
    threads[0].put(wrapper);
    //wait for notification that t has finished.
    synchronized(thisGroup)
    { try{thisGroup.wait();}
      catch(InterruptedException e){return;}
    }
  }

  //cause all threads to terminate. The programmer is responsible
  //for ensuring that the computation is complete.
  public void cancel()
  { for(int i=0; i!= groupSize; i++)
    { threads[i].put(poison); }
  }

  public TaskRunner[] getRunners(){return threads;}
}
```

图5-46 TaskRunnerGroup类，该类初始化并把它管理线程池中的线程

该方法的作用是启动计算。类似的方法是必要的，这是因为新Task只能在另一个Task内部调用fork和join方法产生，而不能由主线程或其他非TaskRunner线程产生。这主要有两点原因：首先fork和join方法需要与执行任务的TaskRunner线程交互（例如，fork方法含有向任务队列添加任务的功能）；其次我们使用Thread.get CurrentThread获得合适的TaskRunner，这样只能在TaskRunner线程所执行的代码中调用fork和join方法。

我们通常希望程序能够创建初始任务并等待，直到任务完成才继续执行。为了实现这个

功能，也为了响应在 task 中调用 fork 的限制，我们定义并实现了一个“wrapper”任务。它的功能是启动初始任务并等待它的完成，最后通知主线程（调用 executeAndWait 的线程）。“wrapper”任务被添加到线程 0 的任务队列中，使它满足被执行的条件并等待它通知我们（利用 notifyAll）它已经完成了。

通过在斐波纳契数字示例中使用 fork、join 和 executeAndWait，我们将更加清晰地了解这些方法。

5. 示例

**计算斐波纳契数字**。图 5-47 和图 5-48 中的代码使用了分布式队列包[⊖]。考虑如下公式：

```
public class Fib extends Task
{
  volatile int number; // number holds value to compute initially,
                       //after computation is replaced by answer
  Fib(int n) { number = n; } //task constructor, initializes number

  //behavior of task
  public void run() {
    int n = number;

    // Handle base cases:
    if (n <= 1) { // Do nothing: fib(0) = 0; fib(1) = 1 }
    // Use sequential code for small problems:
    else if (n <= sequentialThreshold) {
      number = seqFib(n);
    }
    // Otherwise use recursive parallel decomposition:
    else {
      // Construct subtasks:
      Fib f1 = new Fib(n - 1);
      Fib f2 = new Fib(n - 2);

      // Run them in parallel:
      f1.fork();f2.fork();
      // Await completion;
      f1.join();f2.join();

      // Combine results:
      number = f1.number + f2.number;
      // (We know numbers are ready, so directly access them.)
    }
  }

  // Sequential version for arguments less than threshold
  static int seqFib(int n) {
    if (n <= 1) return n;
    else return seqFib(n-1) + seqFib(n-2);
  }

  //method to retrieve answer after checking to make sure
  //computation has finished, note that done and isDone are
  //inherited from the Task class. done is set by the executing
  //(TaskRunner) thread when the run method is finished.
  int getAnswer() {
    if (!isDone()) throw new Error("Not yet computed");
    return number;
  }
/* continued in next figure */
```

图 5-47 计算斐波纳契数字的程序（续见图 5-48）

⊖ 这段代码实质上与 FJTask 包演示的计算斐波纳契数字的类相同，但为了使用前面描述的类，做了一些小修改。

```
/* continued from previous figure */
  //Performance-tuning constant, sequential algorithm is used to
  //find Fibonacci numbers for values <= this threshold
  static int sequentialThreshold = 0;

  public static void main(String[] args) {
      int procs; //number of threads
      int num; //Fibonacci number to compute
      try {
        //read parameters from command line
        procs = Integer.parseInt(args[0]);
        num = Integer.parseInt(args[1]);
        if (args.length > 2)
         sequentialThreshold = Integer.parseInt(args[2]);
      }
      catch (Exception e) {
        System.out.println("Usage: java Fib <threads> <number> "+
            "[<sequentialThreshold>]");
        return;
      }

      //initialize thread pool
      TaskRunnerGroup g = new TaskRunnerGroup(procs);

      //create first task
      Fib f = new Fib(num);

      //execute it
      g.executeAndWait(f);

      //computation has finished, shutdown thread pool
      g.cancel();

      //show result
      long result;
      {result = f.getAnswer();}
      System.out.println("Fib: Size: " + num + " Answer: " + result);
  }

}
```

图 5-48 计算斐波纳契数字的程序（续图 5-47）

$$Fib(0)=0 \tag{5-7}$$

$$Fib(1)=1 \tag{5-8}$$

$$Fib(n+2)=Fib(n)+Fib(n+1) \tag{5-9}$$

这是一个典型的分治算法。为使用任务包，定义了 Fib 类，该类扩展了 Task 类。每一个 Fib 任务都包含一个成员 number，该成员初始化为需要进行斐波纳契计算的数字，随后被替换为计算结果。getAnswer 方法返回计算结果。因为 number 变量将被多个线程访问，所以它声明为 volatile 类型。

run 方法定义每个任务的行为。递归并行分解的过程如下：首先为每个子任务创建一个新的 Fib 对象，并调用每个子任务的 fork 方法，以启动它们的计算；然后为每个子任务调用 join 方法，以等待子任务的完成；最后计算它们结果的总和。

main 方法驱动计算。它首先读取 proc（需要创建的线程数目）、num（进行斐波纳契计算的数字）和可选的 sequentialThreshhold。最后一个是可选参数（默认为 0），用于确定问题大小，决定到底使用串行算法还是并行算法来求解这个问题。获得这些参数

后，main 方法创建一个包含指定数目线程的 TaskRunnerGroup，然后创建一个 Fib 对象，并使用 num 参数初始化。计算开始时，首先将 Fib 对象传递给 TaskRunnerGroup 的 invokeAndWait 方法。当它返回时，计算完成。此时，线程池被 TaskRunnerGroup 的 cancel 方法所关闭。最后，从 Fib 对象中获得结果并进行演示。

6. 相关模式

*共享队列*模式是*共享数据*模式的一个实例。它常常在使用*主 / 从*模式的算法中表示任务队列。它也能够支持基于线程池实现*派生 / 聚合*模式。

注意，当任务队列中的任务被映射到一个连续的整数序列时，用一个单调的共享计数器替换队列，可能会更加高效。

## 5.10 分布式数组模式

1. 问题

常常需要在多个 UE 间划分数组。如何对数组进行划分，才能使最终程序既可读又有高效呢？

2. 背景

在科学计算问题中，大型数组是基础数据结构。微分方程是许多计算问题的核心，而求解这些方程需要使用大型数组。当连续域被离散值集合所替代时，很自然地就需要使用大型数组。在信号处理、静态分析、全局优化和大量其他问题中也需要使用大型数组。因此，高效地处理大型数组是一个重要的问题。

188 ~ 198

如果并行计算机具有足够大的地址空间，能够容纳整个数组，并提供从任何 PE 到任意数组元素的等时访问，那么数组的处理方式是没有必要关心的。但是处理器性能要比内存子系统快得多，连接节点的网络比内存总线慢得多。因此，最终系统的访问时间常常差别很大，具体依赖于哪一个 PE 访问哪一个数组元素。

高效处理大数组所面临的挑战是如何组织数组，使得计算过程中，每个 UE 所需要的元素在恰当的时间恰好位于该 UE 附近。换句话说，数组必须分布在计算机中，使得数组分布匹配计算流。

这种模式对于任何使用大型数组的并行算法都非常重要。特别是算法结构使用几何分解模式、程序结构使用 SPMD 模式时，尤其重要。尽管这种模式在某种程度上与分布式内存环境（在分布式内存环境中，全局数据结构必须分配在所有 PE 上）特别相关，但是如果在一个 NUMA 平台上（在 NUMA 平台中，所有的 PE 可访问全部内存位置，但访问时间不同）实现单个地址空间，是可以应用这种模式的某些思想的。对于这样的平台，不需要显式地分解和分配数组，但是管理内存层次仍然很重要，其目标是使数组元素尽可能接近于需要它们的 PE[⊖]。基于这个原因，在 NUMA 机器中，MPI 程序有时比使用 NUMA 机器的一种本地多线程 API 所实现的相似算法要好得多。进一步讲，这种模式的思想可应用于多线程 API，以保证内存页尽可能接近即将处理它的处理器。例如，当目标系统使用“first touch”页管理策略时，如果每一个数组元素被即将处理它的 PE 初始化，将显著提高系统效率。但是，如果在计

⊖ NUMA 计算机通常由一些将处理器和内存系统子集捆绑在一起的硬件模块构成。在一个硬件模块内部，处理器与自己的内存子系统离得非常近，处理器访问这个内存的时间要比访问一个远程内存所需要的时间少很多。

算的过程中需要重新映射数组，则这种方法将失效。

3. 面临的问题

- **负载均衡**。直到所有的 UE 都完成其工作，并行计算才能完成。所以在 UE 间分配计算负载时，必须尽量保证每一个 UE 的计算时间尽可能接近。
- **高效内存管理**。现代计算机的微处理器性能比内存要快得多。为此，高性能计算机系统通常具有复杂的内存层次。只有充分利用这种内存层次，才有可能实现高性能。实现这个目标的方法是确保处理器和它所要处理的数据尽可能接近。（即缓存中数据的高重用性，并且处理器保持对所需要页的可访问性）。
- **抽象清晰性**。如果数组在 UE 间的划分方式和到本地数组的映射方式是清晰的，则包含分布式数组的程序更容易编写、调试和维护。

199

4. 解决方案

**概述**。以较高的层次描述这个解决方案是比较简单的，但具体细节比较复杂。其基本思想是：将全局数组划分为多个子块，并将这些子块映射到 UE 上。这种映射方式应尽量做到负载均衡，即当计算展开时，尽量保证每一个 UE 具有相等的工作量。除非所有 UE 共享单个地址空间，否则每一个 UE 所要处理的数据都要存储在该 UE 的局部数组中。代码将使用局部数组的索引访问分布式数组的元素。但是，问题和解决方案的算法描述是基于全局数组索引的。在这种情况下，必须清楚全局数组和本地数组间的映射与转换机制：即如何使用全局索引来访问元素；如何使用局部索引和 UE 标识符的组合来访问元素。清晰的映射和转换机制，将是高效使用这种模式所面临的一个挑战。

**数组分布**。数组分布方法已经成为标准。

- **一维（1D）块**。仅在一个维度上分解数组，并且为每个 UE 分配一个子块。例如，对于一个 2D 矩阵，对应于将部分连续行或列分配给每个 UE。这种分配方法有时称为列块或行块分配法（具体依赖于在哪个维度上进行分配）。此时，UE 在概念上组织为一个 lD 数组。
- **二维（2D）块**。和 1D 块一样，为每个 UE 上分配一个子块。但这个数据子块是初始全局数组的一个矩形子块。这种映射将 UE 集合看作一个 2D 数组。
- **块循环**。数组被分解为多个块（使用 lD 或 2D 划分），并且块的数目远大于 UE 数目。然后像分发纸牌一样将这些块循环分配给 UE。此时，UE 可以看作 lD 数组或 2D 数组。

下面将更详细地介绍这些分配方法。为便于演示，如图 5-49 所示[⊖]，我们使用一个秩为 8 的方阵 $A$。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ | $a_{0,4}$ | $a_{0,5}$ | $a_{0,6}$ | $a_{0,7}$ |
| $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ | $a_{1,4}$ | $a_{1,5}$ | $a_{1,6}$ | $a_{1,7}$ |
| $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ | $a_{2,4}$ | $a_{2,5}$ | $a_{2,6}$ | $a_{2,7}$ |
| $a_{3,0}$ | $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ | $a_{3,4}$ | $a_{3,5}$ | $a_{3,6}$ | $a_{3,7}$ |
| $a_{4,0}$ | $a_{4,1}$ | $a_{4,2}$ | $a_{4,3}$ | $a_{4,4}$ | $a_{4,5}$ | $a_{4,6}$ | $a_{4,7}$ |
| $a_{5,0}$ | $a_{5,1}$ | $a_{5,2}$ | $a_{5,3}$ | $a_{5,4}$ | $a_{5,5}$ | $a_{5,6}$ | $a_{5,7}$ |
| $a_{6,0}$ | $a_{6,1}$ | $a_{6,2}$ | $a_{6,3}$ | $a_{6,4}$ | $a_{6,5}$ | $a_{6,6}$ | $a_{6,7}$ |
| $a_{7,0}$ | $a_{7,1}$ | $a_{7,2}$ | $a_{7,3}$ | $a_{7,4}$ | $a_{7,5}$ | $a_{7,6}$ | $a_{7,7}$ |

图 5-49 初始方阵 $A$

⊖ 在本模式的这个图和其他图中，将使用如下符号约定：1）一个矩阵元素表示为一个具有下标的小写字母，其中下标表示索引。例如，$a_{1,2}$ 表示矩阵 $A$ 第 1 行、第 2 列的元素。2）一个子矩阵将表示为一个具有下标的大写字母，其中下标表示索引。例如，$A_{00}$ 表示一个包含 $A$ 的左上角的子矩阵。3）当讨论将 $A$ 的某部分分配给 UE 时，使用 UE 加位于圆括号中的索引来表示不同 UE；例如，如果我们将所有 UE 看作一个 lD 数组，则 $UE(0)$ 表示概念上最左边的 UE；如果我们将所有 UE 看作一个 2D 数组，则 $UE(0,0)$ 表示概念上位于左上角的 UE。假设所有的索引基于 0（即最小索引是 0）。

**1D 块**。图 5-50 演示了 ***A*** 的列块分配：将矩阵 ***A*** 分配到由 4 个 UE 组成的线性数组上。该矩阵仅沿着列索引分解；矩阵的秩除以 UE 总数得到每个块的列数 *MB*（这里是 2）。把矩阵元素 $(i, j)$ 分配给 $UE(j\backslash MB)$ ⊖。

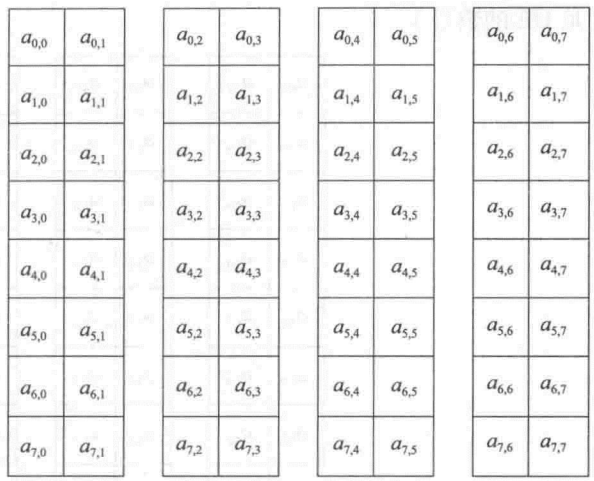



图 5-50 矩阵 ***A*** 到 4 个 UE 的 1D 分配

**UE 映射**。考虑一般情况，假设有一个 $N \times M$ 矩阵，列数目不被 UE 数目 $P$ 整除。在这种情况下，*MB* 是映射到 UE 上的最大列数目，除了 $UE(P\text{-}1)$ 之外，所有的 UE 包含 *MB* 块。于是，$MB=\lceil M/P \rceil$，列 $j$ 的元素被映射到 $UE(\lfloor j/MP \rfloor)$ 上⊜（这归约了前面示例中给出的公式，因为如果 $P$ 能够整除 $M$，$\lceil M/P \rceil = M/P$，并且 $\lfloor j/MB \rfloor = j/MB$）。类似的公式适用于行分配方法。

**局部索引映射**。将列映射到 UE 后，还需要实现全局索引到局部索引的映射。在这个示例中，矩阵元素 $(i,j)$ 映射到局部元素为 $(i, j \bmod MB)$。给定局部索引 $(x,y)$ 和 $UE(w)$，就可以获得全局索引 $(x, wMB+y)$。同理，类似的公式也适用于行分配方法。

**2D 块**。图 5-51 展示了矩阵 ***A*** 到 UE 的 2D 块（大小为 $2 \times 2$）分配。***A*** 沿两维分解，对于每个子块来说，列数等于矩阵的秩除以 UE 矩阵的列数，行数等于矩阵的秩除以 UE 矩阵的行数。把矩阵元素 $(i,j)$ 分配给 $UE(i/2, j/2)$。

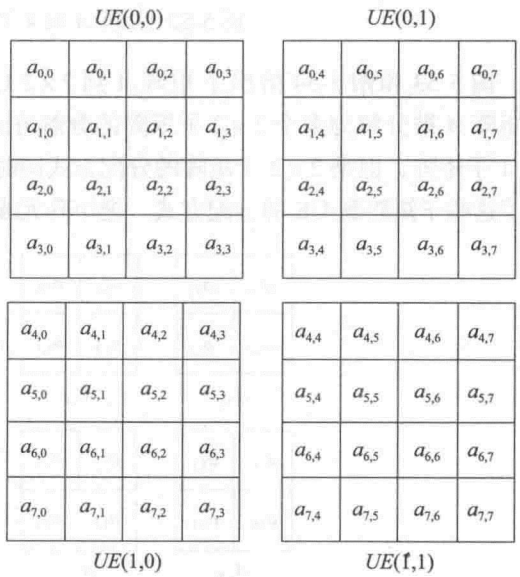



图 5-51 矩阵 ***A*** 到 4 个 UE 的 2D 分配

UE 映射。更常见的是，将 $N \times M$ 矩阵映射到 $P_R \times P_C$ 的 UE 矩阵。一个子块的最大尺寸为 $NB \times MB$，其中 $NB=\lceil N/P_R \rceil$ $MB=\lceil M/P_C \rceil$。全局矩阵元素 $(i,j)$ 映射到 $UE(\lfloor i/NB \rfloor, \lfloor j/MB \rfloor)$ 上。

*局部索引映射*。全局索引 $(i,j)$ 映射到局部索引 $(i \bmod NB, j \bmod MB)$。给定 $UE(z,w)$ 上的局部索引 $(x,y)$，对应的全局索引为 $(zNB+x, wMB+y)$。

**块循环**。块循环的主要思想是创建大大超出 UE 数量的块，并以一种循环方式进行分配

---

⊖ 我们将使用符号“\”表示整数除法，而“/”表示普通除法。这样 $a\backslash b=\lfloor a/b \rfloor$。另外，$\lceil x \rceil$ 表示小于等于 $x$ 的最大整数，$\lfloor x \rfloor$ 表示大于等于 $x$ 的最小整数。例如 $\lceil 4/3 \rceil=1$，$\lfloor 4/2 \rfloor=2$。

⊜ 注意，当 UE 数量不能整除列数时，这并不是在 UE 间分配列的唯一可行方式。还有一种更复杂的分配方法，在某些情况下可实现更好的负载均衡。该方法首先将每个 UE 分配的最小列数定义为 $\lfloor M/P \rfloor$，然后将前 $(M \bmod P)$ 个 UE 分配的列数加 1。例如，对于 $M=10$ 和 $P=4$ 的情况，$UE(0)$ 和 $UE(1)$ 将分配 3 列数据，$UE(2)$ 和 $UE(3)$ 将分配两列数据。

（与分发纸牌类似）。图 5-52 给出了矩阵 $A$ 到长度为 4 的 UE 数组的 1D 块循环分配方式，并演示了矩阵列是如何以循环模式分配给 UE 的。把矩阵元素 $(i,j)$ 分配给 UE($j$ mod 4)（其中 4 是 UE 的数目）。

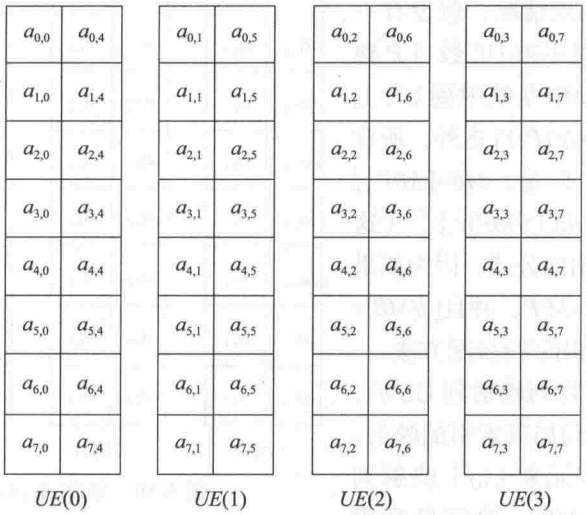



图 5-52 矩阵 $A$ 到 4 个 UE 的 1D 块循环分配

图 5-53 和图 5-54 给出了矩阵 $A$ 到 $2\times2$ UE 数组的 2D 块循环分配方式。图 5-53 演示了矩阵 $A$ 被分解为多个 $2\times2$ 子矩阵的分解方式（我们可以选择一种不同的分解方式，例如 $1\times1$ 子矩阵，但是 $2\times2$ 子矩阵的分配方式同时具有块分配和循环分配两种特征）。图 5-54 给出了这些子矩阵到 UE 的分配方式。把矩阵元素 $(i,j)$ 分配给 $UE(i \bmod 2, j \bmod 2)$。

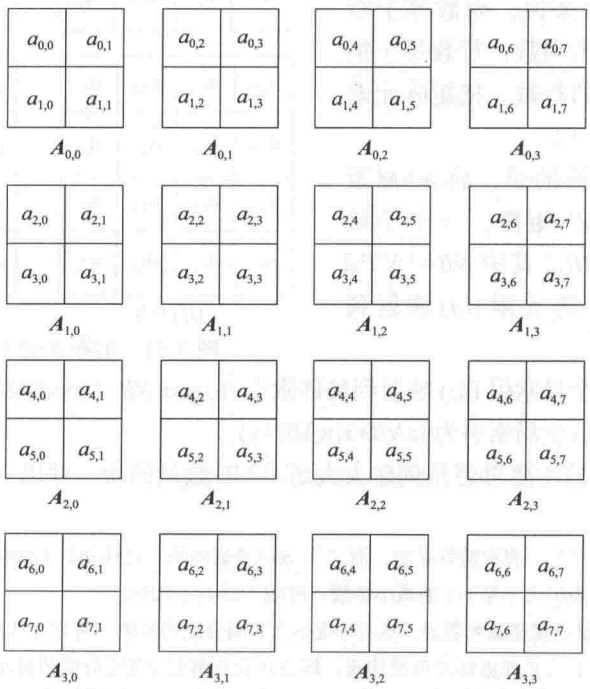



图 5-53 矩阵 $A$ 到 4 个 UE 的 2D 块循环分配，部分 1：分解矩阵 $A$

UE映射。一般情况下，将$N\times M$矩阵映射到$P_R\times P_C$的UE矩阵，子块的大小为$NB\times MB$。把全局矩阵元素$(i,j)$映射到$UE(z,w)$，其中$z=\lfloor i/NB\rfloor \bmod P_R$，$w=\lfloor j/MB\rfloor \bmod P_C$。

局部索引映射。因为把多个块映射到相同的UE上，所以可以从局部索引块或元素角度实现局部索引映射。

从块的角度来看，映射到UE上的每个元素可通过块索引$(l,m)$和块内部索引$(x,y)$得到局部索引。假设全局矩阵元素$(i,j)$映射到某个UE的局部$(l,m)$块的$(x,y)$位置处，其中$(l,m)=(\lfloor i/P_RNB\rfloor,\lfloor j/P_CNB\rfloor)$，$(x,y)=(i \bmod NB, j \bmod MB)$。图5-55演示了映射到$UE(0,0)$的元素的局部索引情况。

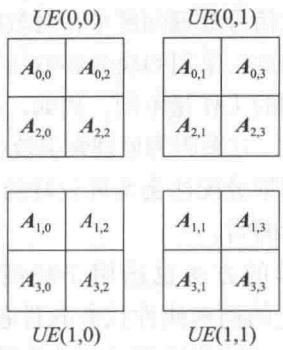



图5-54 矩阵$A$到4个$UE$的2D块循环分配，部分2：将子矩阵分配给UE

例如，全局矩阵元素$a_{5,1}$，当$P_R=P_C=NB=MB=2$时，把该元素映射到$UE(0,0)$上。有4个$2\times 2$块映射到该UE上。从图5-55中可以看到，这个元素位于左下角的块上，即块$LA_{1,0}$上。实际上，根据公式我们可以得到$(l,m)=(\lfloor 5/(2*2)\rfloor,\lfloor 1/(2*2)\rfloor)=(1,0)$。块内局部索引为$(x,y)=(5 \bmod 2, 1 \bmod 2)=(1,1)$。

从元素角度看（每个UE的所有块组成一个连续矩阵），全局索引$(i,j)$映射到的局部索引为$(lNB+x,mMB+y)$，其中，1和$m$的意义与前面相同。图5-56展示了映射到UE(0,0)中的元素。

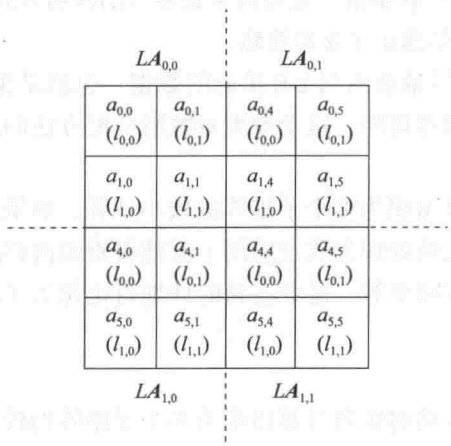



图5-55 矩阵$A$到4个UE的2D块循环分配：分配给$UE(0,0)$的矩阵$A$元素。$LA_{l,m}$是索引为$(l,m)$的子块，每个元素的初始全局索引为$a_{i,j}$，块$LA_{l,m}$内索引为$l_{x,y}$

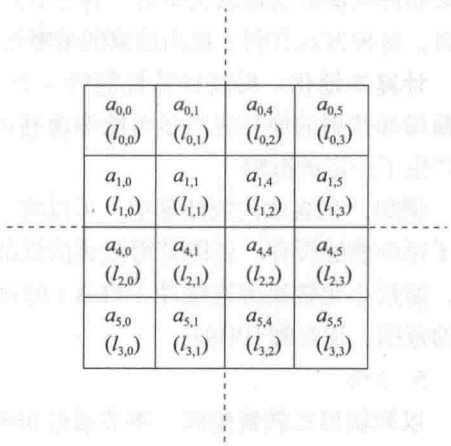



图5-56 矩阵$A$到4个UE的2D块循环分配：从分配给$UE(0,0)$的矩阵$A$元素的角度出发。每个元素通过初始全局索引$a_{i,j}$和它的局部索引$l_{x',y'}$来标识。局部索引为该元素在存储分配给该UE的所有块的连续矩阵上的索引

再次以全局矩阵元素$a_{5,1}$为例，将该数据看作单个矩阵时，该元素的局部索引为$(1\times 2+1,0\times 2+1)=(3,1)$。$UE(z,w)$中块$(l,m)$内的局部索引$(x,y)$对应的全局索引为$(lP_R+z)NB+x,(mP_C+w)MB+y)$。

**分配法的选择**。根据UE上计算负载随计算进行的变化，选择合适的分配算法。例如，在单频道信号处理问题中，对数组的每列执行相同操作，计算量不会随着计算的进行而发生变化。因此，采用列块分解方法可以编写逻辑清晰的代码并能实现较好的负载均衡。如果处理每一列的工作量不同，例如，列索引越大，所需工作量就越多，则列块分解将导致较差的负载均衡。这是因为处理索引较小列的UE将先于处理索引较大列的UE完成工作。在这个示例中，周期分配法会实现较好的负载平衡，因为每个UE既会处理索引较小的列，也会处理索引较大的列。

同样的方法也适用于较高维数组。ScaLAPACK[Sca、BCC+97]（运行在分布式内存计算机上的稠密线性代数软件包）采用2D块周期分配法。我们以高斯消元（一个常用的ScaLAPACK例程）算法讨论选择这种分配方法的原因。该算法也称为LU分解：将稠密方阵转换为一对三角矩阵，即上三角矩阵 $U$ 和下三角矩阵 $L$。该算法从全局矩阵的左上角开始，沿对角线依次处理矩阵元素（消除对角线下面的元素，根据需要将剩余的块转换到右边）。当计算沿对角线进行时，块分配法将导致某些UE空闲。但当使用2D块周期分配法（如图5-54所示）时，每个UE同时包含算法中早期和后期使用的元素，因此能够实现良好的负载均衡。

200
~
206

**映射索引**。前面讨论的示例演示了初始（全局）数组的每个元素到UE的映射方式，以及全局数组中的每个元素在分配后，如何被全局索引集、UE标识符和局部信息的组合所标识。初始问题通常通过全局索引进行描述，但是每个UE内的计算必须通过局部索引进行。为了高效应用这种模式，需要全局索引以及UE和局部索引的组合信息这两者间的关系尽可能地透明。将索引映射隐藏在代码中是非常容易的，但会使程序难以调试。一种更好的方法是使用宏和内联函数实现索引映射，程序员只需了解宏和内联函数，就可以掌握索引的映射方式。同时，这种方式有利于提高抽象的清晰性。后面的示例演示了这种策略。

**计算本地化**。提高计算性能的一个主要原则是尽量重用与UE接近的数据。也就是说，更新局部数据的循环应当尽可能多地利用每一次存储器访问。这个原则对数组分配方法的选择产生了一定的影响。

例如，在线性代数计算中，可以将一个矩阵计算组织为多个子矩阵的较小计算。如果这些子矩阵能够缓存，就能获得显著的性能提升。相似的处理方式也应用于其他级别的内存层次，如最小化转换旁视缓冲（TLB）的命中率和页故障率等。这个主题的详细讨论超出了本书的范围。请参阅[PH98]。

5. 示例

**以列块形式转置矩阵**。本节通过矩阵转置算法说明将矩阵计算组织为多个子矩阵的较小计算的方法，本示例采用方阵，并按照列块方式进行分配。为简化问题，假设UE数目能够整除列数目，因此所有列块的大小相同。如图5-57所示，将矩从阵逻辑上分解为多个方形子矩阵。图5-57中每一个方形子矩阵都用标号进行标识，标号说明了转置后的子块与初始子块之间的关系（例如，标号为 $(A_{0,1})^{\mathrm{T}}$ 在的子块是初始矩阵中标号为 $A_{0,1}$ 子块的转置）。整个算法分为多个阶段，阶段数依赖于UE中子矩阵的数目（也依赖于UE数目）。在第一阶段，对矩阵 $A$ 对角线上的子矩阵进行转置（每个UE转置一个子矩阵），UE间不需要通信。接下来的阶段中，每个UE从对角线下方的第一个子矩阵开始，依次进行转置操作。在这些阶段中，UE必须首先转置其中一个子矩阵，然后将它发送给另外一个UE，并接收一个子矩阵。例如，$UE(1)$ 首先对子矩阵 $(2,1)A^{\mathrm{T}}$ 进行转置；然后将其发送给 $UE(2)$，并从UE(0)接收 $(1,0)A^{\mathrm{T}}$。

图 5-58 和图 5-59 给出了该算法的代码。假设这些块是连续分配的，其中为每个 UE 分配一个列块。因为这部分代码将作为一个较大程序的一部分，所以假设在调用这部分代码之前，数组已经分配。

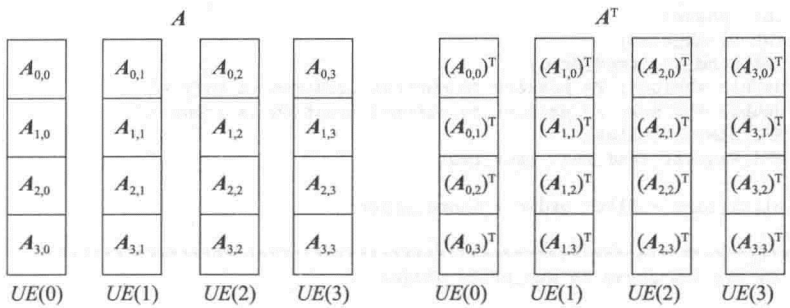



图 5-57 矩阵 $A$ 在 4 个 UE 间的分配及转置

```
/*********************************************************************

NAME: trans_isend_ircv

PURPOSE: This function uses MPI Isend and Irecv to transpose
        a column-block distributed matrix.

*********************************************************************/

#include "mpi.h"
#include <stdio.h>

/*********************************************************************
** This function transposes a local block of a matrix. We don't
** display the text of this function as it is not relevant to the
** point of this example.
*********************************************************************/
void transpose(
   double* A, int Acols, /* input matrix */
   double* B, int Bcols, /* transposed mat */
   int sub_rows, int sub_cols); /* size of slice to transpose */

/*********************************************************************
** Define macros to compute process source and destinations and
** local indices
*********************************************************************/
#define TO(ID, PHASE, NPROC) ((ID + PHASE ) % NPROC)
#define FROM(ID, PHASE, NPROC) ((ID + NPROC - PHASE) % NPROC)
#define BLOCK(BUFF, ID) (BUFF + (ID * block_size))
/* continued in next figure */
```

图 5-58 矩阵转置代码（续见图 5-59）

程序将每个局部列块（一个用于存放 $A$，另一个用于存放转置后的结果）表示为 1D 数组。每个数组由 Num_procs 个子矩阵组成，每个子矩阵的大小 block_size=Block_order*Block_order，其中 Block_order 是每个 UE 分配的列数。因此可以使用 BLOCK 宏来找到索引为 ID 的块：

```
#define BLOCK(BUFF, ID) (BUFF + (ID * block_size))
```

```
/* continued from previous figure */

void trans_isnd_ircv(double *buff, double *trans, int Block_order,
                     double *work, int my_ID, int num_procs)
{
   int iphase;
   int block_size;
   int send_to, recv_from;
   double *bblock; /* pointer to current location in buff */
   double *tblock; /* pointer to current location in trans */
   MPI_Status status;
   MPI_Request send_req, recv_req;

   block_size = Block_order * Block_order;

/*********************************************************************
** Do the transpose in num_procs phases.
**
** In the first phase, do the diagonal block. Then move out
** from the diagonal copying the local matrix into a communication
** buffer (while doing the local transpose) and send to process
** (diag+phase)%num_procs.
*********************************************************************/
   bblock = BLOCK(buff, my_ID);
   tblock = BLOCK(trans, my_ID);

   transpose(bblock, Block_order, tblock, Block_order,
             Block_order, Block_order);

   for (iphase=1; iphase<num_procs; iphase++){
      recv_from = FROM(my_ID, iphase, num_procs);
      tblock = BLOCK(trans, recv_from);
      MPI_Irecv (tblock, block_size, MPI_DOUBLE, recv_from,
            iphase, MPI_COMM_WORLD, &recv_req);

      send_to = TO(my_ID, iphase, num_procs);
      bblock = BLOCK(buff, send_to);
      transpose(bblock, Block_order, work, Block_order,
            Block_order, Block_order);
      MPI_Isend (work, block_size, MPI_DOUBLE, send_to,
               iphase, MPI_COMM_WORLD, &send_req);

      MPI_Wait(&recv_req, &status);
      MPI_Wait(&send_req, &status);
   }
}
```

图 5-59 矩阵转置代码（续图 5-58）

BUFF 是 ID 数组的起始地址（buff 用于存放初始数组，trans 用于存放转置后的数组），ID 是块的第二个索引。例如，可以利用下面的代码寻找两个数组中位于对角线上的块：

```
bblock = BLOCK(buff, my_ID);
tblock = BLOCK(trans, my_ID);
```

在算法的后续阶段，必须确定两件事：应当转置和发送的块的索引；应当接收的块的索引。我们使用宏 TO 和 FROM 完成该任务：

```
#define TO(ID, PHASE, NPROC) ((ID + PHASE ) % NPROC)
#define FROM(ID, PHASE, NPROC) ((ID + NPROC - PHASE) % NPROC)
```

TO 索引表示沿对角线块向下的处理进度：从对角线块开始，依次向下处理，当到达矩阵底部时，折回到矩阵顶部继续处理。在算法的每一个阶段，我们计算哪一个 UE 将接收该块，

然后更新指向该即将被发送块的局部指针（bblock）：

```
send_to = TO(my_ID, iphase, num_procs);
bblock = BLOCK(buff, send_to);
```

相似地，我们计算下一块的来源和对应的局部索引：

```
recv_from = FROM(my_ID, iphase, num_procs);
tblock = BLOCK(trans, recv_from);
```

这个过程持续下去，直到所有的块都转置完毕。

本例使用非阻塞发送（MPI_Isend）和接收（MPI_Irecv）操作（附录B更详细地描述这些原语）。在每个阶段，UE首先调用MPI_Irecv函数，然后再对它即将发送的块进行转置操作，转置完成后，将该块发送到应当接收它的UE中。在循环底部（进行下一阶段之前），程序调用几个MPI_Wait函数，强迫UE等待，直到发送和接收都完成为止。这种方法允许通信和计算的重叠。更重要的是（因为在这个示例中，不存在大量的计算和通信重叠），防止死锁的出现。一种使用规则发送和接收操作的更简单方法是：首先转置将被发送的块，然后发送它，最后（等待）从其他UE接收一个块。但是，当被发送的块很大时，规则发送可能会被阻塞。这是因为没有用于存储消息的足够缓冲空间。在这个示例中，这样的阻塞能够产生死锁。通过使用非阻塞的发送和接收操作，并首先调用非阻塞接收操作，避免了这种情形。

207
~
210

**知名应用**。这种模式广泛应用于科学计算。著名的ScaLAPACK包[Sca、BCC+97]大量使用2D块周期分配法，其文档给出了这种分配方法的映射和索引的详细解释。

在HPF语言[HPF97]的定义中嵌入了几种不同的数组分配方法。

在量子化学中，特别是在post Hartree Fock计算领域中，可以找到这种模式的某些最具创造性的应用。在post Hartree Fock算法中，专门创建了全局数组（GA）包[NHL94、NHL96、NHK+02、Gloa]，以解决分布式数组问题。在[NHL96、LDSH95]中描述了一种更新的方法。

PLAPACK包[ABE+97、PLA、vdG97]采用了一种不同的方法进行数组分配。虽然PLAPACK主要考虑如何根据数组的组织方式使用向量，而不是如何分配数组。但是，根据这些分布式向量，可派生出对应的数组分配。在许多问题中，这些向量对应于问题领域中的物理量，因此PLAPACK小组称此为基于物理的分配法。

6. 相关模式

分布式数组模式常常与几何分解模式和SPMD模式一起使用。

## 5.11 其他支持结构

这门模式语言（以及支持结构模式）是基于OpenMP、MPI与Java程序员在共享内存和分布式内存MIMD计算机上编写代码的惯例。在大多数情形中，并行应用程序程序员将在这门模式语言中找到他们需要的模式。

但是，存在一些其他模式（具有自己的支持结构），在某些情况下，这些模式对于并行编程来说是非常重要的。虽然很少使用，但很有必要了解它们。这些模式为寻找和挖掘并行应用程序的并发性提供见解。随着并行体系结构的不断演化，这些模式所建议的并行编程技术很可能变得非常重要。

本节将简短地描述其中几种额外模式及其支持结构：SIMD、MPMD、客户端-服务器和声明性编程（declarative programming）。我们以问题求解环境的简短讨论来结束本节，这些不是模式，但它们有助于程序员在一个目标问题集中的工作。

### 5.11.1 SIMD

在SIMD计算机中，单指令流作用在多个数据流上。这些机器的灵感源于一个共识，即程序员将发现管理多个指令流是非常困难的，以至于无法完成。许多重要的问题是数据并行
211 的；即可以根据对问题数据领域的并发更新来表达并发性。根据它的逻辑极限，SIMD方法假设可以根据数据来表达所有并行性。然后程序将具有单线程语义，使得对程序的理解和调试变得非常容易。可将SIMD模式的基本思想概括如下。

- 定义一个由虚拟PE组成的网络，这些PE将被映射到实际PE上，通过明确定义的拓扑结构将这些虚拟PE连接起来。理想情况下，该拓扑结构具有两个特征：它与物理机器中的PE的连接方式具有很好的匹配；对于所求解问题所隐含的通信模式是有效的。
- 根据一些数组或其他规则数据结构来表达问题，而且能够利用单个指令流来并发更新这些数组或数据结构。
- 将这些数组与虚拟PE的局部内存相关联。
- 创建单个指令流，该指令流操作一些分块的规则数据结构。这些指令可以具有一个相关的掩码，使得它们能有选择地跳过某些数组元素子集。这对于处理边界条件和其他约束来说，是非常关键的。

对于一个真正的数据并行问题，这种模式非常高效，程序的编写和调试也相对容易[DKK90]。

遗憾的是，大多数数据问题包含一些不是数据并行的子问题。建立核心数据结构，处理边界条件，以及在一个核心数据并行算法之后的后处理，都能够引入不是严格数据并行的逻辑。此外，这种类型的编程与支持数据并行编程的编译器紧密相关。这些编译器的编写困难，导致所产生的代码难于优化，因为它可能与一个程序在一个特定机器上的运行方式相差很远。这样，除了少数用于信号处理应用的专用机器外，这类并行编程及依照SIMD概念所构建的机器已经大量消失。

与SIMD模式最紧密相关的编程环境是高性能Fortran(HPF) [HPF97]。HPF是Fortran90中基于数组构造的一种扩展。HPF的出现是为了支持SIMD机器上的可移植并行编程，且允许SIMD编程模型适用于MIMD计算机。这需要显式地控制数据在PE中的位置，并且可以在计算期间重新映射数据。但是，对严格的数据并行SIMD模型的依赖性，决定了HPF难以用于复杂的应用。最后的大型HPF用户团体位于日本[ZJS+02]，他们已经扩展了这种语言，以放宽了一些数据并行约束[HPF99]。

### 5.11.2 MPMD

顾名思义，当不同程序运行在不同UE上时，在并行算法中使用多程序多数据（MPMD）
212 模式，基本方法如下所示。

- 将问题分解为一个子问题集合，其中每一个子问题映射到一个 UE 子集上。通常每一个 UE 子集对应于另一个并行计算机上的一些节点。
- 创建一些独立的程序来求解适当子问题，并与相关的目标 UE 协调。
- 当需要时，协调运行于各个 UE 上的程序，通常通过一种消息传递框架实现。

在许多方面，MPMD 方法与一个使用 MPI 的 SPMD 程序相差不大。事实上，MPI 最常见的两种实现（MPICH [MPI] 和 LAM/MPI [LAM]）的运行时环境都支持简单的 MPMD 编程。

MPMD 模式的应用通常源于以下两种情形之一。第一，UE 的体系结构可能差别很大，以至无法在整个系统上使用单个程序。例如，当在某种使用多种类型的高性能计算体系结构的计算网格 [Glob、FK03] 中使用并行计算时。当完全不同的模拟程序被组合为一个相互耦合的模拟时为第二种（并且从平行算法的角度更有趣的）情形。

例如，气候是由大气及海洋现象之间的复杂的相互作用形成的。单一的海洋或大气模拟程序已经开发及优化了多年，且模型很容易理解。虽然 SPMD 程序可以直接实现海洋和大气之间的耦合模型，但是一种更有效的方法是采用独立的、经过验证的海洋及大气模拟方案，然后通过某种中间层将两者进行耦合，从而由容易理解的组件模型得到一种新的耦合模型。

尽管 MPICH 和 LAM/MPI 均对 MPMD 编程提供一定支持，但是它们不允许不同的 MPI 实现间的交互，因此仅支持那些使用一种通用 MPI 实现的 MPMD 程序。为了解决不同体系结构和不同 MPI 实现上的更大范围的 MPMD 问题，出现了一种称为可互操作 MPI（iMPI）的新标准。对于 MPI 和 iMPI 来说，利用信息交换来协作 UE 的思想是通用的，但是只在 iMPI 中进行了详细的语义扩展，以解决运行在差别很大的体系结构上的程序所带来的独特挑战。这种多体系结构问题将显著增加通信开销，因此依赖于 iMPI 性能的算法部分必须是相对粗粒度的。

MPMD 程序极罕见。但随着复杂的耦合模拟越来越重要，MPMD 模式的使用将会增加。当网格技术变得更成熟和更广泛部署时，MPMD 模式的使用也将增加。 213

### 5.11.3 客户端 – 服务器计算

客户端 – 服务器体系结构与 MPMD 相关。传统上，这些系统由两层或三层组成，其中前端是在客户计算机上执行的图形用户界面，一个主机后端（通常具有多个处理器）提供了对一个数据库的访问。中间层（如果存在）分配从客户到（可能是多个）后端的请求。Web 服务器是一个熟悉的客户端 – 服务器系统示例。更通用的情形是，服务器可以为客户提供大量的服务，系统的本质方面是服务具有定义完善的接口。并行性可以出现在服务器端（以使得能够同时服务许多客户，或者能够使用并行处理以为单个请求更快速地获得结果）和客户端（以使得能够同时初始化对多个服务器的请求）。

在异构系统中，客户端 - 服务器中所使用的技术特别重要。中间件（例如 CORBA [COR]）提供了一种服务接口规范标准，使得通过组合现有服务来编排新程序，即使在差别很大的硬件平台中提供这些服务并采用不同的编程语言实现这些服务。CORBA 也提供了定位服务的设备。J2EE（Java 2 平台，企业版）[Javb] 也对客户端 – 服务器应用程序提供了显著的支持。在这两种情形中，可互操作性是所面临的主要设计问题。

传统上，客户端－服务器体系结构用于商业领域，而不是科学应用中。广泛地应用于科学计算的网格技术借鉴了客户端－服务器技术，并通过模糊客户端与服务器端之间的区别对其进行扩展。一个网格中的所有资源，无论它们是计算机、仪器、文件系统，还是其他连接到网络中的资源，都是对等的且可以作为客户端和服务器。中间件提供了基于标准的接口，用于将分布在多个可管理域的资源捆绑成单个系统。

### 5.11.4 使用声明语言的并发编程

绝大多数编程采用命令式语言（imperative language）完成，例如，C++、Java或者Fortran。特别是对于传统的科学和工程应用程序来说，更是这样。但是，人工智能社区和少数理论计算机科学家已经开发并成功演示了一种不同类型的语言，声明语言（declarative language）。在这类语言中，程序员描述了一个问题、一个问题域和一些必须满足的条件。然后与该语言相关联的运行时系统使用这些内容寻找有效的解决方案。

声明语义强制了一种不同类型的编程方式，该方式涵盖了本模式语言中讨论的一些方法，但也存在一些不同。声明语言有两种重要类型：函数式语言和逻辑编程语言。

逻辑编程语言基于逻辑推论的形式规则。到目前为止，最常用的逻辑编程语言是Prolog [SS94]，它是基于第一阶谓词运算的编程语言。当扩展Prolog以支持并行性表达时，结果是
214 一种并发逻辑编程语言。在这些Prolog扩展中，采用以下三种方式之一来挖掘并发性："与"并行性（执行多个谓词），"或"并行性（执行多个防护（guard）），或者显式地映射通过一些单赋值变量链接在一起的谓词 [CG861]。

在20世纪80年代末期和90年代早期，并发逻辑编程语言是一个热门的研究领域。它们最终都失败了，因为大多数程序员钟情于更传统的命令式语言。即使声明语义具有很多优点，逻辑编程对于符号推理具有很大的价值，但学习这些语言的艰难性阻止了它们的发展。

较老的和更确定类型的声明编程语言基于函数式编程模式 [Hud89]。LISP是最老的和最著名的函数式语言。在纯函数式语言中，函数中不存在副作用。因此只要能获得输入数据，就能够执行该函数。最终算法根据程序中的数据流来表达并发性，因此最终得到"数据流"算法 [Jag96]。

最著名的并发函数式语言是Sisal[FC090]、Concurrent ML [Rep99、Con]（对ML的一种扩展）和Haskell [HPF]。因为数学表达式可采用函数符号自然地表示，所以Sisal对于处理科学和工程应用程序来说是简单明了的，已经证明它在并行编程方面非常高效。但是，正像逻辑编程语言一样，程序员不希望放弃他们熟悉的命令式语言，因此Sisal最终失败了。尽管Concurrent ML和Haskell在函数式编程团体中都很流行，但它们并没有在高性能计算领域获得显著进展。

### 5.11.5 问题求解环境

如果不提到问题求解环境（PSE），将无法全面讨论并行算法支持结构。PSE是一种针对特定类型问题需求的编程环境。当应用于并行计算时，PSE的隐含意义也是一种特定的算法结构。

PSE的动机是向应用程序员屏蔽并行系统的底层细节。例如，PETsc（用于科学计算的

可移植、可伸缩工具包）[BGMS98] 支持大量的分布式数据结构和使用它们求解偏微分方程（通常用于求解适合几何分解模式的问题）所需要的函数。程序员需要理解 PETSc 中的数据结构，但不必掌握如何高效和可移植地实现的细节。其他重要的 PSE 是 PLAPACK [ABE+97]（用于稠密线性几何问题）和 POOMA [RHC+96]（一种用于科学计算的面向对象框架）。

PSE 并未得到广泛接受。PETSc 很可能是广泛用于应用编程的仅有 PSE。由于它们所面向的问题领域比较窄，因此 PSE 限制了潜在的用户，很难获得广泛的推广。我们相信，随着时间的推移，并行算法的核心模式将变得更易于理解，PSE 将能够扩大它们的影响，在并行编程中成为一支重要的力量。

第 6 章
Patterns for Parallel Programming

# “实现机制”设计空间

前几章已经详细介绍了如何设计算法和用于组织并行程序的高层次结构等内容。本章将介绍的内容主要包括程序源代码和并行程序低层次操作。

对于并行编程首先要确定低层操作及其实现机制，虽然计算机指令集可被高级编程语言调用，但它并没有区分串行程序和并行程序。我们所关心的是并行编程中独特的实现机制。我们将详细介绍并行编程的这些“构造块”。幸运的是，大多数并行程序员仅需要使用这些机制中一个核心子集。核心实现机制分为 3 类：

- UE 管理
- 同步
- 通信

本章介绍了这 3 类最常用的机制。实现机制设计空间的框图和它在编程模式中的位置如图 6-1 所示。

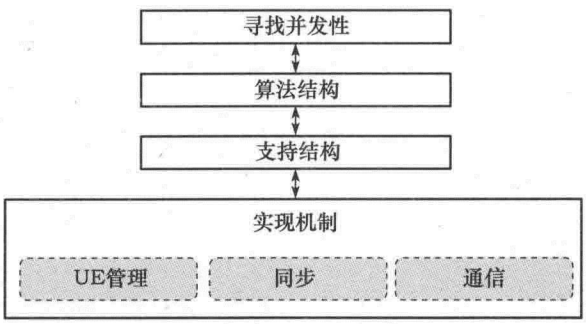



图 6-1　实现机制设计空间的总体框图和它在编程模式中的位置

本章将忽略并行编程模式的具体形式。由于大多数并行编程环境均内置常用的实现机制，因此本章不赘述这些模式的使用方式，仅对每种实现机制进行高层次描述并讨论每种机制如何映射到三种目标编程环境：OpenMP、MPI 和 Java。这种映射在某些情况下比较琐碎，比 API 或一门语言所包含的构造需要更多的知识。特别是在对某种编程模型中包含而在另一些编程模型没有的一些操作进行介绍时该讨论将更有趣。例如，在 OpenMP 中实现消息传递是可行的，虽然并不完美，但它能够工作，并且有时非常有用。

216

本书假定读者已熟悉 OpenMP、MPI 和 Java，以及利用它们编写并行程序的方式。本章将介绍这些 API 的一些特征，使用它们的详细信息见附录。

## 6.1　引言

并行编程通过将指令映射到多个 UE 上来挖掘并发性。每个并行程序必须实现：创建 UE 集合；管理 UE 之间的交互和对共享资源的访问；在 UE 之间交换信息；以某种顺序关闭 UE。这些需求将实现机制分成 3 类。

- **UE 管理**。创建、销毁和管理并行计算中使用的进程与线程。
- **同步**。对不同 UE 中的事件强制约定某种顺序，确保当 UE 集合访问共享资源时，不论如何调度这些 UE，程序都能够正确执行。
- **通信**。在 UE 之间交换信息。

## 6.2 UE 管理

回顾执行单元（或 UE）的定义，UE 是由操作系统为程序员管理并执行计算的实体的抽象。在现代并行编程环境中，存在两种 UE：进程和线程。

进程是一种重量级对象，它包括定义在系统中的位置所需要的状态或上下文，包括内存、程序计数器、寄存器、缓冲区、打开的文件和在操作系统内定义它的上下文所需要的任何内容。在许多系统中，不同进程可以属于不同用户，这样进程就得到了保护。创建新进程和在 217
进程之间交换信息的开销很大，因为所有状态都必须保存和还原。即使是在同一台机器中的进程间通信，由于必须越过保护边界，因此其开销也是非常昂贵的。

线程是一种轻量级 UE。一个进程中包含一个线程集合。属于进程的大多数资源，包括内存等，都由进程内线程共享，因此创建新线程和进行线程间上下文切换的开销较低，仅需要保存程序计数器和某些寄存器。属于同一个进程的线程间通信开销也较低，因为可以通过访问共享内存来完成它。

这两种类型 UE 的管理机制是完全不同的，我们将分别在下面两节介绍。

### 6.2.1 线程的创建 / 销毁

创建线程需要一定的机器周期。根据需要程序员可以在程序中合理地创建和销毁线程，只要不在密集循环或时间关键的内核（time-critical kernel）中进行这些操作，程序整体运行时间将不会受到太大影响。因此，大多数并行编程环境提供一些支持线程创建和销毁的 API。

**OpenMP：线程的创建 / 销毁。**在 OpenMP 中，创建线程的并行指令为：

```
#pragma omp parallel
{ structured block }
```

每个线程独立执行（代码块）structured block 中的指令。代码块是在顶部具有单入口点和在底部具有单出口点的语句块。

OpenMP 中创建线程的数目要么由操作系统控制，要么由程序员控制（详见附录 A）。

线程的销毁发生在代码块的末尾。线程到达代码块的尾部时先处于等待状态，当所有线程到达之后一起被销毁，初始线程或主线程则继续执行。

**Java：线程的创建 / 销毁。**在 Java 中，线程是 `java.lang.Thread` 类的实例或者 `Thread` 类的子类。通常使用关键字 `new` 初始化 `Thread` 对象，然后调用 `start` 方法启动相应线程。这样创建的线程可以访问根据 Java 的作用域规则可见的所有变量。 218

可通过两种方法来声明一个线程的行为。第一种方法是创建 `Thread` 的子类并重写 `run` 方法。下面的代码演示了如何采用这种方法创建一个线程，当该线程启动后，它将执行 `thread_body`。

```
class MyThread extends Thread
{ public void run(){ thread_body }}
```

为了创建并触发线程，需要编写如下代码：

```
Thread t = new MyThread(); //create thread object
t.start(); //launch the thread
```

第二种方法需要定义一个类，实现 `java.lang.Runnable` 接口，该接口只包含一个

方法 public void run()，并传递 Runnable 类的一个实例给 Thread 构造函数。例如，首先定义 Runnable 类：

```
class MyRunnable implements Runnable
{ public void run(){ thread_body }}
```

为了创建并执行线程，创建 Runnable 对象并将它传递给 Thread 构造函数。然后使用 start 方法启动线程：

```
Thread t = new Thread(new MyRunnable()); //create Runnable
                                         //and Thread objects
t.start(); //start the thread
```

在大多数情况下，第二种方法⊖更为常用。

当 run 方法返回时线程终止。与任何其他 Java 程序中的对象一样，Thread 对象在终止后会被垃圾回收器回收。

java.util.concurrent 包定义 Executor 和 ExecutorServices 接口，并提供
219 实现它们的几个类，这些类直接支持高级结构，如主 / 从模式，在 Runnable 的执行过程中隐藏线程的创建和调度等细节。更详细的信息见附录 C。

**MPI：线程的创建 / 销毁。**MPI 基本上是基于进程的，但 MPI 2.0 API[Mesa] 定义了不同级别的线程安全，并提供了在运行时查询系统所支持的线程安全级别的功能，因此 MPI 也是线程可识别的。但 MPI 不具有创建和销毁线程的功能。要在 MPI 程序中使用线程，除了 MPI 之外，程序员必须使用基于线程的编程模型。

### 6.2.2 进程的创建 / 销毁

进程包含在操作系统中定义其位置的所有信息。除了程序计数器和寄存器之外，进程还包含存储空间（即地址空间）、系统缓冲区和定义操作系统中其状态所需的任何信息。因此，进程创建和销毁开销较高，不经常执行此类操作。

**MPI：进程的创建 / 销毁。**在更早的消息传递 API，例如 PVM [Sun90] 中，把创建新进程的功能嵌入 API 中，程序员可以使用 PVM_spawn 命令新建进程：

```
PVM_Spawn(node, program-executable)
```

PVM 允许程序员控制一个程序中哪些计算运行在哪一个节点上。但是，在 MPI 1.1 中，这种能力只能由运行环境决定，程序员无法控制。表面上这似乎是种退步，但这样做有两个原因：首先，通过观察发现大量 PVM 程序基于 SPMD 模式，因此将这种模式构建到 MPI 中是有意义的；其次，这种标准能在更广范围的并行体系结构中实现。MPI 研讨会定义 MPI 标准时，大多数 MPP 计算机不能简单地实现 spawn 语句。

作为 MPI 中创建进程的示例，考虑一个执行函数 foo 的 MPI 程序。程序员利用下面的命令在多个（在该示例中是 4 个）处理器上并行执行：

```
mpirun -np 4 foo
```

运行时，系统查询一个包含所有节点名字的标准文件，选出其中 4 个节点，并在每个节

⊖ 在 Java 中，一个类可以实现任意数目的接口，但仅允许扩展一个超类。这样在第一种方法中扩展 Thread 类就意味着定义了 run 方法的类不能扩展一个特定于应用程序的类。

点上启动相同的命令。 220

当运行在并行计算机节点上的程序退出时，进程被销毁。为了保证彻底终止，MPI 程序的最后一条执行语句为：

```
MPI_Finalize()
```

如果异常退出，例如在发生外部中断时退出，则有可能遗留一些子进程，此时系统要负责清除遗留的任何进程。在大型环境中，MPI 程序频繁地创建和退出，所以系统必须采取适当的清除机制，否则将导致过量的负载。

**Java：进程的创建 / 销毁。**一个 Java 运行时（实现 Java 虚拟机规范）的实例通常对应于一个进程，它包含它支持的 Java 程序所创建的所有线程。标准 API 的 `java.lang.Process` 类和 `java.lang.Runtime` 类中包含一些用于创建新进程以及与进程通信和同步的工具。但通常在 Java 程序调用其他非 Java 程序时才使用这些进程，并不是为了实现并行性。事实上，Java 虚拟机规范甚至不要求新的子进程与父进程并发执行。

Java 也可以用于分布式内存机器，此时利用操作系统工具可在每一台机器中启动一个 JVM 实例。

**OpenMP：进程的创建 / 销毁。**OpenMP 利用 API 实现多线程编程。这些线程共享单个进程。OpenMP 没有创建或销毁进程的能力。

将 OpenMP 扩展到分布式内存计算机，即扩展为一个多进程模型是一个的研究热点 [SLGZ99、BB99、Omn]。这些系统通常采用 MPI 技术，并将进程的创建和销毁留给运行环境。

## 6.3 同步

同步是对多个 UE 中发生的事件先后顺序的强制约束。已有大量介绍同步的文献 [And00]。同步非常复杂且易出错，因此大多数程序员通常仅使用少量同步方法。

### 6.3.1 内存同步和围栅

在典型的共享内存多处理器模型中，每个 UE 执行一个指令序列，其中包括大量共享内存读写操作。可以将所有的计算视为一个原子事件序列，其事件交错地来自于不同 UE。如果 221
UE A 写一个内存位置，UE B 读该位置，则 UE B 将得到 UE A 所写的值。

例如，假设 UE A 做某些工作，然后将变量 `done` 设置为真。此时，UE B 执行一个循环：

```
while (!done) {/*do something but don't change done*/}
```

在这个简单模型中，UE A 最终设置变量 `done`。在下一个循环迭代中，UE B 将读取新值，并终止循环。

但是有几种情况会导致错误。首先，由于新值可能位于缓存中而不是内存中，变量 `done` 的值可能并没有被 UE A 写到内存中，或没有被 UE B 从内存中读取。即使在保证缓存一致性的系统中，`done` 也可能（由于编译器优化）存储在寄存器中，因此它对 UE B 不可见。类似地，UE B 可能尝试读取变量并获得一个旧值，或者由于编译器优化多次读取该值。通常，内存系统属性、编译器、指令排序等因素能够协同以影响内存内容（当查看每个 UE 时）的不确定性。

*内存围栅*（fence）是一种同步事件，它保证所有 UE 看到一致的内存内容，即围栅之前

的所有写操作将对围栅之后的读操作可见，这与经典模型所期望的一样，围栅之后的所有读操作将取得不早于围栅之前的最后一次写入的值。

显然，仅当UE之间存在上下文共享时，才存在内存同步问题。因此，当UE是运行在分布式内存环境中的进程时，通常不会产生这一问题。但对于线程而言，在适当的位置放置围栅能够保证具有多个竞态条件的程序之间的正确执行。

内存围栅的显式管理十分复杂且易于出错。幸运的是，尽管程序员需要考虑这一问题，但很少需要显式处理围栅。下面将会介绍，内存围栅通常由高级的同步构造所隐含。

**OpenMP：围栅。**在OpenMP中，利用flush语句定义内存围栅。

```
#pragma omp flush
```

围栅使得所有对主调UE可见的每个变量在计算机的内存中更新。因为保证一致性需要
222 将某些缓存行、所有系统缓冲区和寄存器写入内存，所以flush操作开销较高。另一种开销较低的flush语句可以列举需要更新的变量：

```
#pragma omp flush (flag)
```

因为OpenMP的高级同步构造隐含了所需的flush，所以程序员很少使用flush构造。但当程序员创建自定义同步构造时，flush操作是非常重要的。例如，对同步（pairwise synchronization）中，同步发生在特定的线程对间，而不是整个线程组中。因为OpenMPAPI不直接支持对同步[⊖]，所以当算法需要它时，程序员必须创建对同步。图6-2中的代码演示了如何在OpenMP中使用flush构造安全地实现对同步。

```
#include <omp.h>
#include <stdio.h>
#define MAX 10 // max number of threads

// Functions used in this program: the details of these
// functions are not relevant so we do not include the
// function bodies.
extern int neighbor(int); // return the ID for a thread's neighbor
extern void do_a_whole_bunch(int);
extern void do_more_stuff();

int main() {
   int flag[MAX]; //Define an array of flags one per thread
   int i;

   for(i=0;i<MAX;i++)flag[i] = 0;

#pragma omp parallel shared (flag)
  {
   int ID;
   ID = omp_get_thread_num(); // returns a unique ID for each thread.
```

图6-2 该程序演示了在OpenMP中实现对同步的方式。flush构造至关重要。它强制内存一致，因此使得对flag数组的更新可见。OpenMP语法细节请参阅附录A

⊖ 如果一个程序在整个线程组中使用同步，则同步的工作将与于线程组的大小无关，即使线程组的大小是1。另一方面，如果一个具有对同步的程序运行于单个线程上，则将产生死锁。OpenMP的设计目标是鼓励代码无论运行于一个线程还是多个线程上，都产生等价的结果，即一个称为串行等价性的属性。这样，不具有串行等价性的高级构造（例如，对同步）都被排除在API之外。

```
    do_a_whole_bunch(ID); // Do a whole bunch of work.

    flag[ID] = 1; // signal that this thread has finished its work
    #pragma omp flush (flag) // make sure all the threads have a chance to
                             // see the updated flag value

    while (!flag[neighbor(ID)]){ // wait to see if neighbor is done.
    #pragma omp flush(flag) // required to see any changes to flag
     }

     do_more_stuff(); // call a function that can't safely start until
                      // the neighbor's work is complete
   } // end parallel region
}
```

图 6-2 （续）

在程序中，每个线程具有两个工作块，与线程组中其他线程并发执行。工作由两个函数表示：do_a_whole_bunch() 和 do_more_stuff()。这两个函数的内容并不重要（这里没有给出）。在该例中，直到邻居完成第一个函数（do_a_whole_bunch()）的工作后，一个线程才能够安全地开始第二个函数（do_more_stuff()）的工作。

该程序使用 SPMD 模式。在对同步中，线程通过设置 flag 数组中的值（根据线程 ID 索引）来交换它们的状态。因为数组必须对所有的线程可见，所以需要共享。在 OpenMP 中通过一个串行区域（即先于线程组的创建）中声明该数组来实现该要求。我们利用一条并行指令来创建线程组：

```
#pragma omp parallel shared(flag)
```

当工作完成时，线程将它的 flag 设置为 1，以通知邻居线程，此时必须调用 flush 以确保其他线程能看到更新值：

```
#pragma omp flush (flag)
```

该线程一直等待，直到其邻居完成了 do_a_whole_bunch() 调用，它再调用 do_more_stuff():

```
while (!flag( neighbor(ID))){ // wait to see if neighbor is done.
#pragma omp flush(flag) // required to see any changes to flag
}
```

在 OpenMP 中，flush 操作仅影响对主调线程可见的变量。如果另一线程写一个共享变量，并通过一个 flush 强制它可以被其他线程读取，则读取该变量的线程仍需要执行一个 flush 以 224
确保它读取最新值。因此，while 循环体必须包含一个 flush(flag) 结构，以确保线程看到 flag 数组中的新值。

如前所述，确定何时需要一个 flush 是十分复杂的。多数情况下，flush 已构建进同步结构中。但当标准结构不足并需要用户自定义同步时，需要在适当的位置放置内存围栅。

**Java：围栅。**与 OpenMP 不同，Java 不提供显式的 flush 构造。事实上，Java 内存模型[⊖]不由 flush 操作定义，而是根据关于“可见性和对应于锁操作的顺序”的约束来定义的。虽然

⊖ Java 是率先尝试准确地声明内存模型的语言之一。其初始规范不够精确，不支持某些同步习惯，且不允许某些合理的编译器优化，从而饱受批评。在 [JSRa] 中描述了 Java 2 1.5 的新规范。在本书中，我们假设遵从新规范。

具体细节很复杂，但思想很简单：假设线程1执行某些操作，并同时拥有锁$L$，然后释放锁，同时线程2获取锁$L$，则在线程1释放锁之前，所有写对于线程2（在获取锁之后）是可见的。进一步，当一个线程通过调用start方法启动时，启动线程可看到在调用点对于调用者可见的所有写。相似地，当一个线程调用join时，调用者将看到终止线程执行的所有写。

Java允许变量被声明为volatile类型。当一个变量被标记为volatile时，对该变量的所有写将立即可见，保证所有读获得最后一次写入的值。这样，编译器通过volatile获知内存同步[⊖]。在一个包含段的程序的Java版本中（其中期望done被另外一个线程设置）：

```
while (!done) {.../*do something but don't change done*/}
```

当声明变量done时，将它标记为volatile类型，然后可以忽略程序其他部分与这个变量相关的内存同步问题。

```
volatile boolean done = false;
```

因为volatile关键字仅能用于引用一个数组元素，而不能用于引用整个数组，所以
225 Java 2 1.5中引入的java.util.concurrent.atomic包中添加了原子数组的概念，允许通过volatile语义访问数组的每一个元素。例如，在图6-2所示的OpenMP示例的一个版本中，将flag数组声明为AtomicIntegerArray类型，并利用该类的set与get方法更新与读取它的值。

用于确保适当的内存同步的另一种方法是同步块，如下所示：

```
synchronized(some_object){/*do something with shared variables*/}
```

6.3.3节和附录C将更详细地描述同步块。现在只需理解some_object与锁隐式相关联，编译器产生的代码用于在执行同步块之前获取这个锁，并在退出同步块时释放该锁。这意味着通过确保对变量的所有访问发生在与相同对象相关联的同步块中，即可保证对该变量的访问同步。

**MPI：围栅。**围栅仅存在于共享内存的环境中。在MPI 2.0之前的MPI规范中，不向程序员公开共享内存，因此不需要用户可调用的围栅。但由于MPI 2.0包含单边通信构造，即可以创建对MPI程序中其他进程可见的内存“窗口”，因此数据可以被单个进程写入或读出，而不用显式地与拥有这个内存区域的进程协作。这些内存窗口需要某种类型的围栅，由于在编写本书时还无法获得MPI 2.0的实现，这里不进行讨论。

### 6.3.2 栅栏

*栅栏*（barrier）是一个同步点，UE集合中的每个成员必须都到达该点后才能继续执行。如果某个UE先到达，则它将等待，直到所有其他UE到达之后才继续执行。

栅栏是常用的高级同步构造之一，它既适用于面向进程的环境（如MPI），也适用于基于线程的系统（如OpenMP和Java）。

**MPI：栅栏。**在MPI中通过调用如下函数调用栅栏：

⊖ 读取一个volatile变量所对应的内存同步与获取一个与该变量相关的锁所具有的效果相同，相反，写与释放一个锁具有相同的效果。

MPI_Barrier(MPI_COMM)

其中，MPI_COMM 是一个通信域，定义进程组和通信上下文。该通信域进程组中所有进程均参与栅栏。栅栏对于程序员可能是不可见的，但几乎始终利用消息对的级联实现的，并使用归约实现（参考 6.4.2 节）。 226

图 6-3 展示了一个栅栏示例程序。该程序创建了 MPI 环境，使用 MPI 的时间例程来记录 runit() 函数的执行时间（这里没有给出该函数的代码）。

MPI_Wtime() 227

```
#include <mpi.h> // MPI include file
#include <stdio.h>

extern void runit();

int main(int argc, char **argv) {
    int num_procs; // number of processes in the group
    int ID; // unique identifier ranging from 0 to (num_procs-1)
    double time_init, time_final, time_elapsed;

//
// Initialize MPI and set up the SPMD program
//
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &ID);
    MPI_Comm_size (MPI_COMM_WORLD, &num_procs);

//
// Ensure that all processes are set up and ready to go before timing
// runit()
//
    MPI_Barrier(MPI_COMM_WORLD);

    time_init = MPI_Wtime();

    runit(); // a function that we wish to time on each process

    time_final = MPI_Wtime();

    time_elapsed = time_final - time_init;

    printf(" I am \%d and my computation took \%f seconds\n",
           ID, time_elapsed);

     MPI_Finalize();
     return 0;
}
```

图 6-3 调用一个栅栏的 MPI 程序，用于计时 runit() 函数的执行时间

该函数返回一个双精度值，保存自上一次调用该函数后逝去的时间（以秒为单位）。即两次函数调用返回值的差即为一个函数的执行时间，也称挂钟（wall clock）时间，表示计算机外部的一个时钟所耗去的时间。

MPI 的进程启动或初始化的时间可能相差很大。为了在所有的进程中保持时间的一致性，需要保证所有进程一起进入代码。为了解决这个问题，需要在代码的时间函数部分前放置一个栅栏。

**OpenMP：栅栏**。在 OpenMP 中，利用一条简单的命令设置栅栏：

```
#pragma omp barrier
```

线程组中的所有线程参与这个栅栏。图 6-4 展示了一个栅栏示例，该程序实际上与 MPI 程序相同。栅栏用于确保在进行时间测量之前，所有线程已完成启动活动。时间例程 ompWtick() 类似于 MPI_Wtime()，并采用相同的方式定义。

```
#include <omp.h>
#include <stdio.h>

extern void runit();

int main(int argc, char **argv) {
    double time_init, time_final, time_elapsed;

#pragma omp parallel private(time_init, time_final, time_elapsed)
{
    int ID;
    ID = omp_get_thread_num();
//
// ensure that all threads are set up and ready to go before timing runit()
//
    #pragma omp barrier

    time_init = omp_get_wtime();

    runit(); // a function that we wish to time on each thread

    time_final = omp_get_wtime();

    time_ elapsed = time_final - time_init;

    printf(" I am %d and my computation took \%f seconds\n",
          ID, time_elapsed);
}
return 0;
}
```

图 6-4 调用一个栅栏的 OpenMP 程序，用于计时 runit() 函数的执行时间

除了图 6-4 中所示的显式栅栏外，OpenMP 自动地在工作共享（worksharing）结构（for、single、section 等）的结尾处插入栅栏。但是可以调用 nowait 子句关闭这种隐式栅栏。

栅栏隐含了对 flush 的调用，因此 OpenMP 栅栏也创建内存围栅。这些内存 flush 组成了 UE 在栅栏处等待时所浪费的 CPU 周期，因此十分耗时。栅栏在任何编程环境中开销都非常大，只能用于那些需要确保程序语义正确的地方，如果没有性能原因，应该仅用于绝对必需的地方。

**Java：栅栏。**Java 早期并不包含栅栏原语，尽管使用 Java 中的工具可以简单地创建栅栏，就像在公共域 util.concurrent 包中所做的一样 [Lea]。在 Java 2 1.5 中，java.util.concurrent 包中提供了相似的功能。

CyclicBarrier 与前面介绍的栅栏相似，它包含两个构造函数：一个将栅栏处同步的线程的数目作为参数，另一个将与 Runnable 对象相关联的线程数作为参数，其中 Runnable 对象的 run 方法将被最后到达栅栏的线程所执行。当一个线程到达栅栏时，它调用该栅栏的 await 方法。如果一个线程因为某异常而提前终止，则其他线程将抛出 BrokenBarrierException 异常。当通过栅栏时，CyclicBarrier 对象将自动重置，

并可以多次使用（例如，在一个循环中）。

图 6-5 展示了上一个栅栏示例的 Java 版本，使用 CyclicBarrier。

```
import java.util.concurrent.*;

public class TimerExample implements Runnable {
   static int N;
   static CyclicBarrier barrier;
   final int ID;

   public void run()
  {
    //wait at barrier until all threads are ready
   System.out.println(ID + " at await");
   try{ barrier.await(); }
   catch (InterruptedException ex) { Thread.dumpStack();}
   catch (BrokenBarrierException ex){Thread.dumpStack();}

   //record start time
   long time_init = System.currentTimeMillis();

   //execute function to be timed on each thread
   runit();

   //record ending time
   long time_final = System.currentTimeMillis();

   //print elapsed time
   System.out.println("Elapsed time for thread "+ID+
       " = "+(time_final-time_init)+" msecs");
   return;
   }

   void runit(){...} //definition of runit()

   TimerExample(int ID){this.ID = ID;}

   public static void main(String[] args)
  { N = Integer.parseInt(args[0]); //read number of threads

     barrier = new CyclicBarrier(N); //instantiate barrier

     for(int i = 0; i!= N; i++) //create and start threads
     {new Thread(new TimerExample(i)).start(); }
  }
}
```

图 6-5 一个使用 CyclicBarrier 的 Java 程序，用于计时 runit() 函数的执行时间

java.util.concurrent 包也提供了一个同步原语 CountDownLatch。CountDownLatch 被初始化为某个值 $N$。每一次调用 countDown 方法将该值减 1，之后线程将调用 await 方法，直到该值为 0。countDown（可比喻为“到达栅栏”）与 await（等待其他的线程）相分离比“线程等待所有其他线程到达一个栅栏”应用更广泛。例如，单个线程可以等待 $N$ 个事件发生，或者 $N$ 个线程可以等待单个事件发生。CountDownLatch 只能使用一次，不能重置。附录 C 展示一个使用 CountDownLatch 的例子。

### 6.3.3 互斥

当共享内存或其他系统资源（如文件系统）时，程序必须确保多个 UE 互不干扰。例如，

若两个线程准备在同一时间更新一个共享数据结构，则竞态条件的可能导致该结构产生不一致的状态。临界区是一个可能与其他 UE 执行的语句序列相冲突的语句序列。若两个语句块访问同一个数据，且至少一条语句序列修改了数据，则这两个语句序列会产生冲突。为了保护临界区中的资源，程序员必须使用一种称为互斥（mutual exclusion）的机制以确保每一时刻仅有一个线程将执行临界区中的代码。

当使用互斥时，很可能一个线程继续执行，而一个或多个线程被阻塞，等待进入临界区。在并行程序中，这将严重影响执行效率。因此使用互斥构造时必须非常谨慎，应该尽量最小化互斥所保护的代码量，并尽可能让线程组中的成员交错到达互斥构造，这将减少等待线程的数量。5.8 节介绍了互斥应当保护的对象。这里主要介绍用于保护临界区的实现机制。

**OpenMP：互斥。**在 OpenMP 中，可以利用临界区构造简单地实现互斥，图 6-6 中给出 OpenMP 中使用临界区构造的一个示例。

```
#include <omp.h>
#include <stdio.h>
#define N 1000

extern double big_computation(int, int);
extern void consume_results(int, double, double *);

int main() {
  double global_result[N];

  #pragma omp parallel shared (global_result)
  {
    double local_result;
    int I;
    int ID = omp_get_thread_num(); // set a thread ID

    #pragma omp for
    for(i=0;i<N;i++){
         local_result = big_computation(ID, i); // carry out the UE's work
        #pragma omp critical
       {
            consume_results(ID, local_result, global_result);
       }
    }
  }
  return 0;
}
```

图 6-6 一个包含临界区的 OpenMP 程序

图 6-6 所示程序创建一个线程组，它们协作实现对 big_computation() 进行的 $N$ 个调用。下面的指令是一个 OpenMP 构造，它告诉编译器将循环分配到线程组中：

```
#pragma omp for
```

在 big_computation() 执行完毕后，结果需要组合到全局数据结构中，以保存结果。

虽然这里没有给出代码，假设 consume_results() 的更新能够以任意顺序进行，但执行过程必须是交替的，即一个线程更新完后另一个线程才能更新。critical 指令提供这一功能。第一个完成 big_computation() 的线程进入封装的代码块，并调用 consume_results()。如果一个线程到达临界区，而另外一个线程正在临界区内，则它将等待，直到临界区内的线程完成。

临界区是一种高成本的同步操作。在临界区入口，一个线程刷新所有可见变量，以确保在临界区中看到一致的内存内容。在临界区的末尾，为使临界区中的任何内存更新对于线程组中的其他线程可见，需要对线程所有可见变量进行第二次刷新。

临界区构造不仅开销大，而且并不通用。它不能用于一个线程组中的线程子集，也不适用于不同代码块间的互斥。为此，OpenMP API 为互斥提供了一种级别较低并且更灵活的构造，称为锁。

不同共享内存的 API 中的锁基本相似。程序员声明一个锁并对其初始化。一次仅允许有一个线程拥有这个锁，其他试图获取锁的线程将被阻塞。阻塞以等待一个锁并不高效，因此许多锁 API 允许线程仅仅测试一个锁的可用性而不是获取它。这样，一个线程可以执行其他指令，然后再尝试获取锁。

图 6-7 展示了一个使用 OpenMP 锁的示例，确保一次只有一个线程写入标准输出。

```
#include <omp.h>
#include <stdio.h>

   int main() {
   omp_lock_t lock; // declare the lock using the lock
                    // type defined in omp.h

   omp_set_num_threads(5);
   omp_init_lock (&lock); // initialize the lock
   #pragma omp parallel shared (lock)
   {
       int id = omp_get_thread_num();
       omp_set_lock (&lock);
       printf("\n only thread %d can do this print\n",id);
       omp_unset_lock (&lock);
   }
}
```

图 6-7 在 OpenMP 中使用锁的示例

该程序声明锁 lock 为 `omp_lock_t` 类型并对其初始化，这是一个不透明的对象，因此程序员需要通过 OpenMP API 运行时库以安全地利用锁对象，不用考虑锁类型的细节。然后调用 `omp_init_lock` 初始化锁。

228 ~ 232

为了实现并发性管理，锁必须被线程组中所有成员共享。因此，锁在 `parallel` 指令之前定义锁，并且声明为一个共享变量（OpenMP 中变量默认是共享的，但这里显式调用 `shared` 子句以示强调）。在并行区域的内部，一个线程可以设置锁，之后其他尝试设置相同锁的线程被阻塞并等待，直到该锁被释放。

与 OpenMP 临界区不同，OpenMP 锁没有内置内存围栅。如果获得锁的线程执行的操作依赖其他线中程的任何值，则可能会由于内存不一致而导致程序出错。管理内存一致性对于许多程序员来说是非常困难的，因此大多数 OpenMP 程序员更多地使用更安全的临界区而避免使用锁。

**Java：互斥。**Java 语言利用同步块构造实现互斥，在 Java 2 1.5 中，则利用 `java.util.concurrent.lock` 包中新的锁类。在 Java 程序中，每个对象隐式地包含它自己的锁。每个同步块有一个相关联的对象，线程必须获取该对象中的锁后才能执场块体操作。当线程退出同步块体时，无论是正常还是非正常退出（非正常时将抛出一个异常），都会释放锁。

图 6-8 展示了前面示例的 Java 版本。线程执行嵌套 Worker 类中的 run 方法。注意，因为 N 为 final 类型（不可变），所以它可以安全地被任何线程访问，而不需要任何同步。

```
public class Example {
static final int N = 10;
static double[] global_result = new double[N];

  public static void main(String[] args) throws InterruptedException
  {
    //create and start N threads
    Thread[] t = new Thread[N];
    for (int i = 0; i != N; i++)
      {t[i] = new Thread(new Worker(i)); t[i].start(); }

    //wait for all N threads to finish
    for (int i = 0; i != N; i++){t[i].join();}

    //print the results
    for (int i = 0; i!=N; i++)
      {System.out.print(global_result[i] + " ");}
    System.out.println("done");
  }

static class Worker implements Runnable
{
  double local_result;
  int i;
  int ID;

  Worker(int ID){this.ID = ID;}

  //main work of threads
  public void run()
  { //perform the main computation
    local_result = big_computation(ID, i);

    //update global variables in synchronized block
    synchronized(this.getClass())
     {consume_results(ID, local_result, global_result);}
  }

  //define computation
  double big_computation(int ID, int i){ . . . }

  //define result update
  void consume_results(int ID, double local_result,
                       double[] global_result){. . .}
 }
}
```

图 6-8　图 6-6 中 OpenMP 程序的 Java 版本

为了利用同步块实现互斥，它们必须与相同的对象关联。在本例中，run 方法中的同步块使用 this.getClass() 作为参数。该表达式返回一个引用，指向一个表示 Worker 类的运时对象。这是一种确保调用者使用相同对象的简单方式。可以声明 java.lang.Object 的一个全局实例（或者其他类的一个实例），并用它作为同步块的参数。重要的是所有线程同步于同一个对象。使用 this 存在一个潜在的错误，它不强制互斥，因为每个 worker 线程将同步于它自己——这样将锁定一个不同的锁。

这在 Java 多线程程序中是一个常见的误解和错误源，所以需要再次强调，同步锁仅阻止临界区被其他线程访问，并且其冲突语句被封装在一个利用相同对象作为参数的同步块中。

与不同对象关联的同步块不互斥（也不保证内存同步）。而且方法中的一个同步块的出现不会约束非同步块中的代码。因此，忘记需要的同步块或弄错同步块的参数后果非常严重。

图 6-8 中的代码构造方式与 OpenMP 示例类似。在 Java 程序中，更常用的方式是将共享数据结构封装在类中，仅通过同步方法访问它。同步方法是一个包含完整方法的同步块的特例。普通方法隐式地同步于 this，静态方法隐式地同步于该类的对象。这种方法将同步从线程对共享数据结构访问转移到数据结构本身。通常这是一种较好的方法，图 6-9 使用这种方法改写了前面的示例。现在，global_result 变量被封装在 Example2 类中，并被标识为私有类型（Worker 类不能为嵌套类）。worker 访问 global-result 数组的唯一方式是调用 consume_results 方法，即 Example2 类中的一个同步方法。因此，同步的责任已从定义辅助线程的类转移到拥有 global_result 数组的类。

```
public class Example2 {
   static final int N = 10;
   private static double[] global_result = new double[N];

   public static void main(String[] args) throws InterruptedException
   { //create and start N threads
      Thread[] t = new Thread[N];
      for (int i = 0; i != N; i++)
         {t[i] = new Thread(new Worker(i)); t[i].start(); }

      //wait for all threads to terminate
      for (int i = 0; i != N; i++){t[i].join();}

      //print results
      for (int i = 0; i!=N; i++)
         {System.out.print(global_result[i] + " ");}
      System.out.println("done");
   }

   //synchronized method serializing consume_results method
   synchronized static void consume_results(int ID, double local_result)
      { global_result[ID] = . . . }
}

class Worker implements Runnable
{
   double local_result;
   int i;
   int ID;

   Worker(int ID){this.ID = ID;}

   public void run()
   { //perform the main computation
      local_result = big_computation(ID, i); //carry out the UE's work

      //invoke method to update results
      Example2.consume_results(ID, local_result);
   }

   //define computation
   double big_computation(int ID, int i){ . . . }
}
```

图 6-9 利用同步方法实现互斥的 Java 程序

Java 中的同步锁构造存在某些缺陷。对于并行程序而言，在尝试获取锁之前，没有判断该

锁是否可用的方法，也没有方法中断正在等待同步块的线程，同步块强制锁以一种嵌套模式被获取和释放。这将使一些编程风格不被支持，如在一个块中获取锁而在另一个块中释放该锁。

由于这些缺陷，许多程序员已经创建了自己的锁类，以取代 Java 内置的同步块。例如，请参考 [Lea]。对应于这种情形，Java 2 1.5 中的 `java.util.concurrent.locks` 包提供了几种锁类以替代同步块。详细内容请见附录 C。

**MPI：互斥。**与大多数同步构造一样，仅当语句在共享上下文中执行时才需要互斥。因此，对于类似 MPI 这种不共享任何数据的 API，其标准中不支持临界区。考虑图 6-6 中的 OpenMP 程序。若要在 MPI 中实现类似方法，即每次只允许一个 UE 更新数据结构，则典型的方法是指定一个进程完成这种更新，而其他进程将需要把更新的数据发送给该进程。图 6-10 中展示了这种方法。

```
#include <mpi.h> // MPI include file
#include <stdio.h>
#define N 1000
extern void consume_results(int, double, double * );
extern double big_computation(int, int);

int main(int argc, char **argv) {
    int Tag1 = 1; int Tag2 = 2; // message tags
    int num_procs; // number of processes in group
    int ID; // unique identifier from 0 to (num_procs-1)
    double local_result, global_result[N];
    int i, ID_CRIT;
    MPI_Status stat; // MPI status parameter

// Initialize MPI and set up the SPMD program
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &ID);
    MPI_Comm_size (MPI_COMM_WORLD, &num_procs);

    // Need at least two processes for this method to work
    if(num_procs < 2) MPI_Abort(MPI_COMM_WORLD,-1);

// Dedicate the last process to managing update of final result
    ID_CRIT = num_procs-1;
    if (ID == ID_CRIT) {
       int ID_sender; // variable to hold ID of sender
       for(i=0;i<N;i++){
          MPI_Recv(&local_result, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                   MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
          ID_sender = stat.MPI_SOURCE;
          consume_results(ID_sender, local_result, global_result);
       }
    }
    else {
      num_procs--;
      for(i=ID;i<N;i+=num_procs){ // cyclic distribution of loop iterations
        local_result = big_computation(ID, i); // carry out UE's work

// Send local result using a synchronous Send - a send that doesn't
// return until a matching receive has been posted.
        MPI_Ssend (&local_result, 1, MPI_DOUBLE, ID_CRIT, ID,
                   MPI_COMM_WORLD);
      }
    }
    MPI_Finalize();
    return 0;
}
```

图 6-10 具有需要互斥的更新操作的 MPI 程序示例，一个进程用于更新该数据结构

该程序使用了 SPMD 模式。类似图 6-6 中的 OpenMP 程序，它利用一个循环实现 big_computation() 的 N 次调用，把调用的结果放置到一个全局数据结构中，一次仅有一个 UE 中的结果被应用。

选择具有最大编号的 UE 管理临界区，然后该进程执行一个循环，并执行 N 次接收。通过使用 MPI_Recv() 语句中的 MPI_ANY_SOURCE 和 MPI_ANY_TAG 的值，拥有 big_computation() 调用的结果的消息以任意顺序被接收。如果需要标记或 ID，可以从 MPI_Recv() 返回的状态变量中恢复它们。

其他的 UE 实现对 big_computation() 的 N 次调用。因为一个 UE 已经用于管理临界区，所以计算中的有效进程数目减 1。我们使用一种循环迭代的周期分配法，像 5.4 节的示例中所描述的一样。它以一种轮循方式分配循环的迭代。一个 UE 完成它的计算后，把结果发送给管理临界区的进程[⊖]。

## 6.4 通信

UE 执行并行算法时通常需要交换信息。共享内存环境默认提供了这种功能，但在这些系统中所面临的挑战是同步对共享内存的访问，以保证无论 UE 的执行速度和顺序如何变化，结果都是正确的。在分布式内存系统中，因为共享资源很少，所以保护这些资源的显式同步需求相应较少。而通信成为程序员所要面临的主要问题。

233 ~ 237

### 6.4.1 消息传递

消息是最基本的通信元素，由一个包含消息标识（例如，源、目的地和一个标记）的头和一系列二进制位组成。消息传递通常是双边的，即一个消息显式地在一对 UE 之间发送，从某个源到某个目的地。除了直接消息传递外，还有一些通信事件，它们在单个通信事件中包含多个 UE（通常是所有的 UE），被称为集合通信（collective communication）。这些集合通信事件通常也包括计算，例如，计算全局和，即对分布在系统中的一个值集求和，并放置在每一个节点上。

下面几节将更详细地介绍通信机制。首先介绍 MPI、OpenMP 和 Java 中的基本消息传递。然后介绍集合通信，主要讲解 MPI 和 OpenMP 中的归约操作。最后简单地介绍在并行程序中管理通信的其他几种方法。

**MPI：消息传递。**一对 UE 间的消息传递是最基本的通信操作。消息由一个 UE 发送，另一个 UE 接收。在最常规的通信中，发送和接收操作是成对的。

图 6-11 和图 6-12 中的 MPI 程序中，一个处理器环重复地计算一个字段（程序中的 field）内的元素。为简单起见，我们假设更新操作中的依赖性为每一个 UE 仅需要它左边邻居的信息进行更新。

---

⊖ 利用同步发送（MPI_Ssend()），从而尽可能模拟共享内存的互斥行为，同步发送需要等待匹配的 MPI 接收后才返回。但是在分布式内存环境中，由于额外的开销，一般不需要让发送进程等待。通常 MPI 程序员会尽量降低并行开销，仅在必要的时候才使用同步消息传递。在本例中，标准模式消息传递函数 MPI_Send() 和 MPI_Recv() 是更好的选择，除非通信的外部条件要求两个进程满足某种顺序约束，因此强制它们相互同步；或者通信缓冲区或者其他系统资源限制了接收消息的能力，从而强制发送端进程等待，直到接收端准备好。

```
#include <stdio.h>
#include "mpi.h" // MPI include file
#define IS_ODD(x) ((x)%2) // test for an odd int

// prototypes for functions to initialize the problem, extract
// the boundary region to share, and perform the field update.
// The contents of these functions are not provided.

extern void init (int, int, double *, double * , double *);
extern void extract_boundary (int, int, int, int, double *);
extern void update (int, int, int, int, double *, double *);
extern void output_results (int, double *);

int main(int argc, char **argv) {
   int Tag1 = 1; // message tag
   int nprocs; // the number of processes in the group
   int ID; // the process rank
   int Nsize; // Problem size (order of field matrix)
   int Bsize; // Number of doubles in the boundary
   int Nsteps; // Number of iterations
   double *field, *boundary, *incoming;

   int i, left, right;
   MPI_Status stat; // MPI status parameter
//
// Initialize MPI and set up the SPMD program
//
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &ID);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

   init (Nsize, Bsize, field, boundary, incoming);

/* continued in next figure */
```

图 6-11 该 MPI 程序使用一个处理器环和一种通信模式，将信息右移。计算功能不影响通信本身，故未显示（续见图 6-12）

```
/* continued from previous figure */

  // assume a ring of processors and a communication pattern
  // where boundaries are shifted to the right.

  left = (ID+1); if(left>(nprocs-1)) left = 0;
  right = (ID-1); if(right<0)right = nprocs-1;

  for(i = 0; i < Nsteps; i++){

    extract_boundary(Nsize, Bsize, ID, nprocs, boundary);
    if(IS_ODD(ID)){
      MPI_Send (boundary, Bsize, MPI_DOUBLE, right, Tag1,
                MPI_COMM_WORLD);
      MPI_Recv (incoming, Bsize, MPI_DOUBLE, left, Tag1,
                MPI_COMM_WORLD, &stat);
    }
    else {
      MPI_Recv (incoming, Bsize, MPI_DOUBLE, left, Tag1,
                MPI_COMM_WORLD, &stat);
      MPI_Send (boundary, Bsize, MPI_DOUBLE, right, Tag1,
                MPI_COMM_WORLD);
    }
    update(Nsize, Bsize, ID, nprocs, field, incoming);
  }
  output_results(Nsize, field);
  MPI_Finalize();
  return 0;
}
```

图 6-12 该 MPI 程序使用一个处理器环和一种通信模式，将信息右移。计算功能不影响通信本身，故未显示（续图 6-11）

图 6-11 和图 6-12 中只显示了与通信相关的程序部分，实际更新操作字段、初始化字段，以及字段和边界数据的结构均被忽略。

图 6-11 和图 6-12 的程序首先声明变量，然后初始化 MPI 环境。在这种 SPMD 模式中，并行算法中由进程排序 ID 和组中的进程数目所驱动。

初始化 field 后，通过计算 `left` 和 `right` 变量（确定哪些进程分别位于左侧和右侧），来确立通信模式。在本例中，计算非常简单，实现了一个通信环模式。在更复杂的程序中，这些索引计算可能比较复杂、难以理解和易错。

程序的核心循环执行大量步骤。每一步都收集边界数据，并通过一个环连接与邻居通信，然后更新本地字段块。

注意，必须在奇数和偶数进程间显式交换通信，以确保匹配的通信事件在两个程序上连贯地排序。这很重要，因为在缓冲空间有限的系统上，直到相关的接收发送之后，才能返回发送。

**OpenMP：消息传递。**在 OpenMP 中，消息传递是一种高效策略。OpenMP 中消息传递的一种方式是模仿一个 MPI 算法，例如，当把一个 MPI 程序移植到仅支持 OpenMP 的系统中时。另一种情形是使用 NUMA 机器时，其数据的局部性是非常重要的。通过使用 SPMD 模式和消息传递，程序员能够更精确地控制程序的数据与系统的内存层次的结合方式，从而提高性能。

图 6-13 和图 6-14 中展示了在 OpenMP 中使用消息传递的一种简单方式，程序基本上与图 6-11 和图 6-12 中的 MPI 程序相同。与前面一样，程序中隐藏了与并行算法无关的细节。

```
#include <stdio.h>
#include <omp.h> // OpenMP include file
#define MAX 10 // maximum number of threads
//
// prototypes for functions to initialize the problem,
// extract the boundary region to share, and perform the
// field update. Note: the initialize routine is different
// here in that it sets up a large shared array (that is, for
// the full problem), not just a local block.
//
extern void init (int, int, double *, double * , double *);
extern void extract_boundary (int, int, double *, double *);
extern void update (int, int, double *, double *);
extern void output_results (int, double *);

int main(int argc, char **argv) {
    int Nsize; // Problem size (order of field matrix)
    int Bsize; // Number of doubles in the boundary
    double *field;
    double *boundary[MAX]; // array of pointers to a buffer to hold
                                        // boundary data
//
// Create Team of Threads
//

    init (Nsize, Bsize, field, boundary, incoming);
/* continued in next figure */
```

图 6-13 该 OpenMP 程序使用了一个线程环和一种通信模式，该通信模式使信息右移（续见图 6-14）

```
/* continued from previous figure */

#pragma omp parallel shared(boundary, field, Bsize, Nsize)

//
// Set up the SPMD program. Note: by declaring ID and Num_threads
// inside the parallel region, we make them private to each thread.
//
     int ID, nprocs, i, left;

     ID = omp_get_thread_num();
     nprocs = omp_get_num_threads();

     if (nprocs > MAX) {
        exit (-1);
     }

     //
     // assume a ring of processors and a communication pattern
     // where boundaries are shifted to the right.
     //
     left = (ID-1); if(left<0) left = nprocs-1;

     for(i = 0; i < Nsteps; i++){

#pragma omp barrier

        extract_boundary(Nsize, Bsize, ID, nprocs, boundary[ID]);

#pragma omp barrier

        update(Nsize, Bsize, ID, nprocs, field, boundary[left]);
     }
     output_results(Nsize, field);
     return 0;
}
```

图 6-14 该 OpenMP 程序使用了一个线程环和一种通信模式，该通信模式使信息右移（续图 6-13）

在 OpenMP 中，通过从共享数据结构中读取数据来完成消息传递。在 MPI 中，消息传递函数隐式地保证同步，但 OpenMP 中的同步是显式的。第一个 `barrier` 确保所有的线程位于同一个点（即准备填充将共享的 `boundary`）。第二个 `barrier` 确保在接收端使用数据之前，所有的线程已经完成了对它们的 `boundary` 的填充。

如果每一个线程中包含的工作量相近，则这种策略非常高效。而当某些线程比其他的线程运行快（由于硬件负载较小或者数据非均匀分布）时，栅栏将导致较高的并行开销，这是因为某些线程需要在栅栏处等待其他较慢线程。可以通过精细地调节问题的同步需求，例如使用对同步，来缓解这一瓶颈。

图 6-15 中的“消息传递”OpenMP 程序展示了 OpenMP 中引入对同步的一种方式，其代码比图 6-13 和图 6-14 中的程序更复杂。基本思想是添加一个称为 `done` 的数组，线程使用它来表明其缓冲区（即 boundary 数据）是否已经准备好并可使用。一个线程填充它的缓冲区，设置它的标记（flag），然后更新它的变量，以确保其他线程能够看到更新后的值。然后该线程检查它的邻居的标记，并等待（使用一个所谓的*旋转锁*（spin lock）），直到邻居的缓冲区准备好接收数据为止。

```
    int done[MAX]; // an array of flags dimensioned for the
                   // boundary data
//
// Create Team of Threads
//
    init (Nsize, Bsize, field, boundary, incoming);

#pragma omp parallel shared(boundary, field, Bsize, Nsize)

//
// Set up the SPMD program. Note: by declaring ID and Num_threads
// inside the parallel region, we make them private to each thread.
//
    int ID, nprocs, i, left;

    ID = omp_get_thread_num();
    nprocs = omp_get_num_threads();
    if (nprocs > MAX) { exit (-1); }

    //
    // assume a ring of processors and a communication pattern
    // where boundaries are shifted to the right.
    //
    left = (ID-1); if(left<0) left = nprocs-1;

    done[ID] = 0; // set flag stating "buffer ready to fill"
    for(i = 0; i < Nsteps; i++){

    #pragma omp barrier // all visible variables flushed, so we
                        // don't need to flush "done".
      extract_boundary(Nsize, Bsize, ID, num_procs, boundary[ID]);

      done[ID] = 1; // flag that the buffer is ready to use

      #pragma omp flush (done)

      while (done[left] != 1){
        #pragma omp flush (done)
      }

      update(Nsize, Bsize, ID, num_procs, field, boundary[left]);

      done[left] = 0; // set flag stating "buffer ready to fill"
    }
    output_results(Nsize, field);
```

图 6-15 算法同图 6-13 和图 6-14，但具有更细致的同步管理（对同步）

这段代码更加复杂，但是它在两个方面更加高效。首先，前一代码中的栅栏使得所有线程均等待整个线程组。如果任意一个线程因某种原因延迟，则它将延缓整个线程组，特别是当线程间的工作负载不均衡时，会导致较高的性能开销。其次，将两个栅栏替换为一个栅栏和一系列的 flush 操作，虽然 flush 操作的开销较高，但仅更新一个小数组（done），此时对单个数组的多次刷新操作可能比栅栏操作所隐含的对所有线程可见数据的单个刷新操快得多。

**Java：消息传递**。Java 语言没有消息传递的定义（因为它实现利用线程进行并行编程的工具）。可以使用 OpenMP 中所讨论的类似的消息传递技术。但是，Java 的发布版本的标准类库中提供了对分布式环境中各类通信的支持。本节不提供这些技术的示例，而是简要介绍一些工具。

Java 为异构环境中的客户端服务器模式的分布式计算提供了支持，可以在两个 PE 之

间建立一个 TCP 套接字连接，并通过该连接发送和接收数据。相关的类位于 java.net 和 java.io 包中。串行工具（参考 java.io.Serializable 接口）支持将复杂的数据结构转换为一个字节序列，该字节序列可以通过网络传输（或写到一个文件中），并在目的端重新构建。RMI（远程方法调用）包 java.rmi.* 提供了一种机制，使得一个 JVM 上的对象可以调用另一个可能运行在一台不同的机器上的 JVM 上的对象的方法。串行化 RMI 包用于配置方法的参数，并将结果返回给调用者。

尽管 java.io、java.net 和 java.rmi.* 包提供了便利的编程抽象并在它们针对的领域中工作良好，但会产生较高的并行开销，因此不适合高性能计算。高性能计算机通常使用计算机同构网络，因此由 Java 的 TCP/IP 所支持的通用分布式计算方法将导致一些在高性能计算中通常不需要的数据转换和检查。另外一种并行开销来自 Java 所强调的可移植性，这使得 Java 的网络支持设计需要采用能够适应不同平台的工具。一个主要的问题是阻塞 I/O。例如，这意味着套接字上的读操作将被阻塞，直到获得数据，而高性能计算通常通过创建一个新线程来执行读操作，从而避免应用盲等。由于读操作一次仅能应用于一个套接字，所以为每个通信方使用独立的线程是一种不可伸缩的编程方法。

为了优化这些影响高性能企业服务器的缺陷，Java 2 1.4 的 java.nio 包中引入了一些新 I/O 工具。因为 Java 规范已经分成 3 个独立的版本（企业版、核心版和移动版），所以核心版 Java 和企业版 Java API 再也不局限于仅支持那些能够被功能最少的设备所支持的特征。Pugh 和 Sacco[PS04] 所报告的结果表明，一些新工具能提供足够的性能，使 Java 成为集群中高性能计算的一个合理选择。

java.nio 包提供了非阻塞 I/O 和一些选择器，使得一个线程能够监控几个套接字连接。这种机制依赖称为通道（channel）的新抽象，它作为套接字、文件和硬件设备等的开放连接。SocketChanncl 用于 TCP 或 UDP 连接上的通信。程序可以用非阻塞操作将 SocketChannel 中的内容读入到 ByteBuffer 中，或者将 ByteBuffer 中的内容写到 SocketChannel 中。缓冲区是 Java 2 1.4 中引入的另外一种新抽象，是一些容器，用于存放基本类型的线性无限序列，维护一个包含当前位置的状态（以及某些其他信息），并利用 put 和 get 操作访问，put 操作将一个元素放置到当前位置所指定的位置，get 操作从当前位置获取一个元素。缓冲区可以分配为直接的或间接的。直接缓冲区的空间在 JVM 所管理的内存空间的外部，并受垃圾回收限制。因此，直接缓冲区不会被垃圾回收器移动，指向直接缓冲区的引用可以传递给系统级的网络软件，这减少了 JVM 的一个复制步骤。遗憾的是，它的代价是需要更复杂的编程，因为 put 和 get 操作使用起来不是特别方便。但是，通常人们将不在程序的主计算中使用 Buffer，而是将使用一个更便利的数据结构，例如，数组，能够批量地将数组中的数据复制到缓冲区中，或者批量地从 Buffer 中将数据复制到数组中[⊖]。

许多并行程序员期望获得比 Java 中的标准包所支持的更高级或者更熟悉的通信抽象。研究人员已经使用了各种方法为 Java 实现了类似于 MPI 的绑定。一种方法是使用 JNI（Java 本地接口）来绑定现有的 MPI 库。例如，JavaMPI [Min97] 和 mpiJava [BCKL98]。其他方法使用以 Java 编写的新通信系统。这些系统没有广泛使用，处于各种可用性状态。某些系统 [JCS98] 尝试为遵从 [BC00] 中所描述的标准的程序员提供一种类似于 MPI 的体验，而其他

238 ~ 244

⊖ Pugh 和 Sacco [PS04] 已经指出，实际上发送之前在缓冲区和数组之间进行批量复制要比去掉数组并使用 put 和 get 操作直接更新缓冲区要快。

系统试验一些可选择的模型 [Man]。[AMJS02] 中给出了它们的概述。这些系统通常使用老的 java.io，因为 java.nio 包最近才可用。一个例外是 Pugh 和 Sacco [PS04] 所报告的工作。尽管在编写本书时，他们还没有提供可下载的软件，但他们描述了一个包，该包提供了 MPI 的一个子集，该包所具有的性能度量为在集群中进行高性能计算且具有 java.nio 的 Java 描绘了一种乐观的前景。可以期望，在不久的将来将有更多的包构建在 java.nio 之上，人们的并行编程工作将更简单。

### 6.4.2 集合通信

当两个以上的 UE 参与通信事件时，该事件称为集合通信操作（collective communication operation）。作为一个消息传递库，MPI 包含大多数主要的集合通信操作。

- **广播**。发送单条消息给所有 UE。
- **栅栏**。程序中的同步点，所有 UE 都必须先到达该点，然后才能继续向后执行。本章前面已经描述了这一内容，但在 MPI 中这一机制是通过集合通信实现的。
- **归约**。获得一个对象集合，其中每个对象位于一个 UE 上。操作将它们组合为位于一个 UE 之上的单个对象（MPI_Reduce），或者组合它们并将最终结果广播到每个 UE 上（MPI_Allreduce）。

归约是这几种操作中最常用的，在并行算法中占有重要地位，因此也包含在大多数并行编程 API 中（包括共享内存模型 API，如 OpenMP）。本节主要讨论归约的通信模式和算法，这些思路均可用于其他全局通信操作中。

**归约**。*归约*通过重复利用某种二元运算符组合数据集中的数据项对，最终得到单个数据值，通常这种二元运算是可结合和可交换的，包括求和、积或者数组中元素最大值。通常，能将操作表示为如下计算：

$$V_0 \circ V_1 \circ \cdots \circ V_{m-1} \tag{6-1}$$

其中，$\circ$ 是一种二元运算符。实现归约的基本算法即从左到右串行地计算，这种方法并没有潜在的并行度。但是，如果是可结合的，则式（6-1）包含并行度，可以并行计算 $V_0$，$V_1$，…，$V_{m-1}$ 的一些子集的结果，然后再将它们组合在一起。如果 $\circ$ 也是可交换的，则计算不仅能够重组，而且能够重新排序，这为并发执行提供了额外的可行性。

不是所有的归约操作符都具有这两种属性，但是可以考虑该运算符是否能够被当作可结合的和或可交换的，并且不会显著改变计算的结果。例如，浮点加不是严格可结合的（因为数值表示的有限精度会造成舍入误差，特别是两个操作数间的量级差别很大时），但是如果相加的所有数据项具有大致相等的量级，则通常足够接近于可结合的，此时可以利用下面介绍的并行策略。如果数据项的量级差别很大，则将无法利用该并行策略。

大多数并行编程模型都包括归约操作。

- **MPI**。MPI 提供了通用函数 MPI_Reduce 和 MPI_Allreduce，并支持几种常用的归约操作（MPI_MIN、MPI_MAX 和 MAN_SUM）。
- **OpenMP**。包含 reduction 子句，可用于并行区域或一个工作分摊构造，提供了加、减、乘以及大量的位以及逻辑运算符。

图 6-16 展示了一个 MPI 程序，它是前一个归约示例的延续。图 6-3 中的前一示例使用栅栏来强制一致性的计时，即独立地为每个 UE 计时，作为程序负载平衡的指标，通常将测量出最短时间、最长时间和平均时间。图 6-16 的程序通过 MPI_Reduce 的 3 次调用来完成这些任务，即分别调用 MPI_MIN、MPI_MAX 和 MPI_SUM 函数。

```
#include "mpi.h" // MPI include file
#include <stdio.h>

int main(int argc, char**argv) {
    int num_procs; // the number of processes in the group
     int ID; // a unique identifier ranging from 0 to (num_procs-1)
    double local_result;
    double time_init, time_final, time_elapsed;
    double min_time, max_time, ave_time;
//
// Initialize MPI and set up the SPMD program
//
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &ID);
    MPI_Comm_size (MPI_COMM_WORLD, &num_procs);
//
// Ensure that all processes are set up and ready to go before timing
// runit()
//
    MPI_Barrier(MPI_COMM_WORLD);

    time_init = MPI_Wtime();

    runit(); // a function that we wish to time on each process

    time_final = MPI_Wtime();

    time_elapsed = time_final - time_init;

    MPI_Reduce (&time_elapsed, &min_time, 1, MPI_DOUBLE, MPI_MIN, 0,
                                             MPI_COMM_WORLD);
    MPI_Reduce (&time_elapsed, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0,
                                             MPI_COMM_WORLD);
    MPI_Reduce (&time_elapsed, &ave_time, 1, MPI_DOUBLE, MPI_SUM, 0,
                                             MPI_COMM_WORLD);
    if (ID == 0){
       ave_time = ave_time /(double)num_procs;
       printf(" min, ave and max times (secs): %f, %f, %f\n",
              min_time, ave_time,max_time);
    }

     MPI_Finalize();
     return 0;
}
```

图 6-16 用于计时 runit() 的函数执行时间的 MPI 程序，使用 MPI_Rduce 计算最短、最长和平均运行时间

图 6-17 展示了 OpenMP 中的一个归约示例，它是图 6-16 的一个 OpenMP 版本。在并行区域之前声明了线程数目变量（num_threads）和平均时间变量（ave_time），它们在并行区域的内部和后面都是可见的。在 parallel 指令中使用了一个 reduction 子句，表明将对变量 ave_time 做求和归约。编译器将创建该变量的一个局部副本，并将它初始化为问题中运算符的单位元（加法运算的单位元是 0）。每个线程将其局部总和放到这个局部副本中，在并行区的末尾，所有的局部副本被组合进 ave_time 以产生最终值。

```
#include <stdio.h>
#include <omp.h> // OpenMP include file

extern void runit();

int main(int argc, char**argv) {
     int num_threads; // the number of processes in the group
     double ave_time=0.0;

#pragma omp parallel reduction(+ : ave_time) num_threads(10)
{
    double local_result;
    double time_init, time_final, time_elapsed;

// The single construct causes one thread to set the value of
// the num_threads variable. The other threads wait at the
// barrier implied by the single construct.
    #pragma omp single
        num_threads = omp_get_num_threads();

    time_init = omp_get_wtime();

    runit(); // a function that we wish to time on each process

    time_final = omp_get_wtime();

    time_elapsed = time_final - time_init;

    ave_time += time_elapsed;
}
    ave_time = ave_time/(double)num_threads;
    printf(" ave time (secs): %f\n", ave_time);
    return 0;
}
```

图 6-17 用于计时 runit() 的函数执行时间的 OpenMP 程序，使用归约子句计算运行时间总和

要计算平均时间，还需要知道并行区域中的线程个数。确定线程数目的唯一方式是在并行区域中调用 omp_num_thread() 函数。对同一变量的多次写会相互干扰，因此将 omp_num_thread() 的调用放在一个 single 构造中，确保仅有一个线程设置 num_threads 共享变量的值。OpenMP 的 single 块隐含了一个栅栏，因此为了确保所有线程同时进入计时代码，不需要再显式地包含一个栅栏。最后，在图 6-17 中仅计算了平均时间，因为最小和最大运算符没有包括在 C/C++ OpenMP 的 2.0 版本中。

**实现归约操作**。大多数程序员不会实现自己的归约操作，因为大多数并行编程模型中已经实现了归约机制。但是 Java 并不提供归约运算符，因此 Java 程序员需要实现相应的并行归约算法。在任何情况下，用于归约的算法是有帮助的，值得深入理解。

下面将讨论一些基于树的最简单的可伸缩并行归约算法，尽管有些算法不是最优的，但它们揭示了一些更优的归约算法 [CKP+93] 中的大多数问题。

**串行计算**。如果归约运算符不是可结合的，或者在不显著影响结果的情况下不能够被当作可结合的，则需要在单个 UE 中串行地执行整个归约，算法如图 6-18 所示。如果仅有一个 UE 需要归约结果，则最简单的方法是让该 UE 执行该操作；如果所有 UE 需要该结果，则归约操作之后可以调用一个广播操作，将结果广播给其他 UE。为了简化问题，在图 6-18 中，UE 的数目与数据项数目相同。但算法可扩展为处理数据项数目比 UE 数目多的情况，只是仍然需要在单个 UE 中进行所有的实际计算。因为缺乏可结合性，显然，这种解决方案不具备

并发性。这里为了完整起见提到了并发性，因为如果归约操作表示相对少部分计算（其他部分不具有可挖掘的并发性），则并发性可能很有用并比较合适。

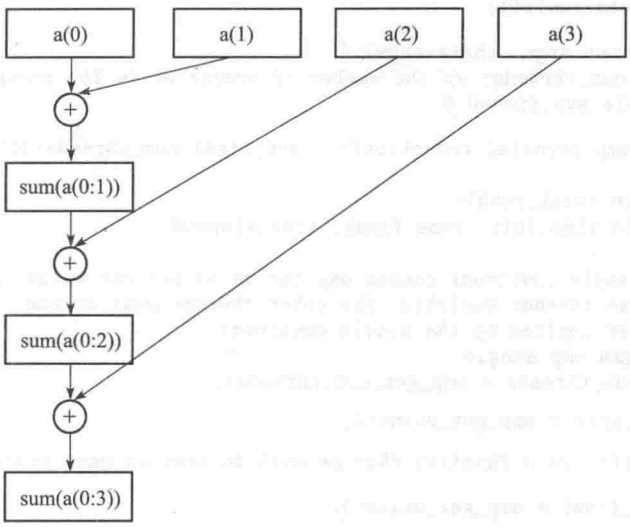



图 6-18 计算 a（0）~ a（3）总和的串行归约算法。Sum（a（i:j））表示数组 a 中 i ~ j 元素的总和

在这种方法中，组合两个元素的操作必须串行执行，因此最简单的方式是所有操作被某个任务执行。但是由于数据依赖性的原因（图 6-18 中的箭头所示），如果归约操作作为一个包含多个并发 UE 的较大计算的一部分执行，则需要非常谨慎；不执行归约的 UE 能够继续其他的工作，但必须保证不影响归约操作的计算（例如，如果多个 UE 共享访问一个数组，并且其中一个 UE 正在计算数组元素的总和，则其他 UE 不应当同时修改数组的元素）。在一个消息传递环境中，这通常可以通过使用消息传递以强制数据依赖性约束来完成（即不执行归约操作的 UE 将它们的数据发送给实际进行计算的某个 UE）。在其他环境中，不执行归约操作的 UE 可以通过一个栅栏强制等待。

**基于树的归约。**如果归约运算符是可结合的或能够被当作可结合的，则可利用图 6-19 中基于树的算法。该算法由一系列的阶段组成，每个阶段都有一半的 UE 将其数据传递给其他 UE。初始时所有 UE 包含在归约中，但每个阶段后一半的 UE 将不进行后续运算，最终结果将在某一 UE 中计算。

为简化起见，在图 6-19 中，UE 数目与数据项数目相同，但算法可以扩展到处理数据项数目比 UE 数目多的情况，其方式是每个 UE 首先对数据项的一个子集执行一个串行归约，然后利用图 6-19 所示的算法进发计算（每个 UE 的串行归约是独立的，能够并发地执行）。

在基于树的归约算法中，某些（不是全部）组合两个元素的操作可以并发地执行（例如，在图 6-19 中，可以并发计算 sum（a（0:1））和 sum（a（2:3）），但 sum（a（0:3））的计算必须等待）。使用 $2^n$ 个 UE 执行基于树的归约算法时，更为通用的策略是划分为 $n$ 个阶段，其中每个阶段所包含的并发操作数目是前一个阶段的一半。在串行阶段，需要满足图 6-19 中所示的数据依赖性。在消息传递环境中，这通常通过合理的消息传递来完成；在其他环境可以通过在每个阶段之后使用栅栏同步来实现。

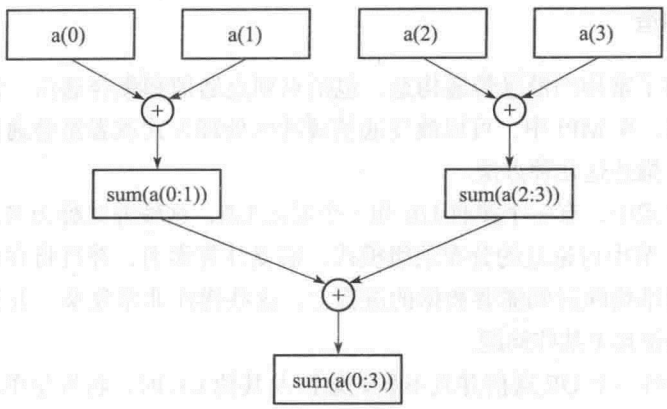



图 6-19 在一个具有 4 个 UE 的系统中计算 a（0）~ a（3）的总和的基于树的归约。Sum（a（i:j））表示数组 a 中 i ~ j 元素的总和

如果只有一个 UE 需要归约结果，则基于树归约算法是最优的。如果其他 UE 也需要最终结果，则归约操作之后可以紧跟一个广播操作，而广播操作可以利用与图 6-19 所示的归约相反的过程实现，即在每一个阶段中获得最终结果的 UE 将值传给另一 UE，这样获得数据的 UE 数目将增加一倍。

**递归加倍（recursive doubling）**。如果所有 UE 都必须知道归约的最终结果，则图 6-20 中的递归加倍策略要优于在基于树的方法后面调用一个广播操作。

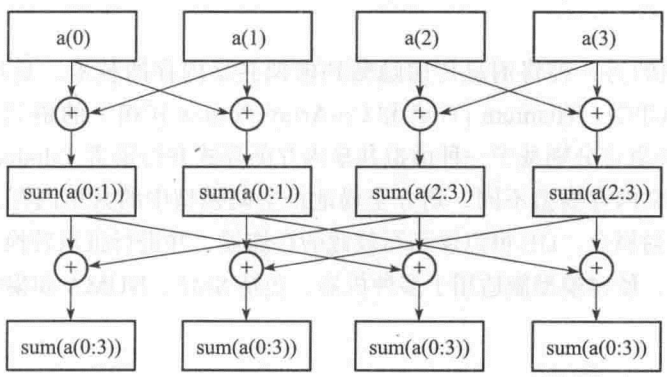



图 6-20 计算 a（0）~ a（3）的总和的递归加倍归约。Sum（a（i:j））表示数组 a 中 i ~ j 元素的总和

与基于树的算法一样，如果 UE 的数目等于 $2^n$，则算法分成 $n$ 个步骤。在算法的开始，每个 UE 有一些值用于归约。这些值在本地组合为单个值以实现归约。第一个阶段中，偶数编号的 UE 与其奇数编号的邻居交换部分和。第二个阶段中，距离为 2 的两个 UE 交换部分和。第三个阶段中，距离为 4 的两个 UE 交换部分和。依此类推，在每个阶段将两个交换部分和的 UE 的距离加倍，直到归约完成。

在 $n$ 个阶段后，每个 UE 都得到归约的最终结果。将这个策略与前面使用基于树的算法后调用一个广播的策略相比，归约和广播都需要 $n$ 个步骤，广播直到归约完成后才能够开始，因此所消耗的时间是 $O(2n)$，在这 $2n$ 个步骤期间，许多 UE 是空闲的。但是，递归加倍算法在每个阶段涉及所有 UE，并且 $n$ 个步骤后，在每个 UE 中产生单个归约值。

### 6.4.3 其他通信构造

前面已经介绍了常用的消息传递构造，包括点到点通信和集合通信。消息传递中有多种重要的变体，例如，在 MPI 中，可以改变通信缓冲区处理方式或者重叠通信和计算从而改进程序性能。附录 B 描述这几种方法。

在前述通信方式中，有一个接收 UE 和一个发送 UE，这种方式称为*双边通信*。对某些算法，例如，在 5.10 节中讨论过的分布数组模式，需要计算索引，并且将存放在分布式数据结构中，然后将数据结构映射到需要数据的进程上，这些操作非常复杂，并且难以调试，可以利用单边通信来避免其中某些问题。

245
~
251

当一个 UE 与另一个 UE 通信并且不显式地涉及其他 UE 时，将发生单边通信。例如，消息可以直接被写入目的节点的缓冲区中，而且不用涉及接收 UE。MPI 2.0 标准包括一个单边通信 API。单边通信的早期实现是一个称为 GA 或 Globa Arrays[NHL94、NHL96、NHK+02、Gloa] 的通信系统。GA 提供了一个简单的单边通信环境，主要服务于分布式数组算法。

另一种方法是将显式通信替换为虚拟共享内存，物理内存可以是分布式的，因此使用“虚拟”一词。20 世纪 90 年代早期非常流行的方法是 Linda [CG91]。Linda 是基于一种称为元组空间的相连虚拟共享内存。Linda 中的操作包括 put、take 或 read 一个数值集，并把它捆绑到一个称为元组的对象中。通过匹配一个模板来访问元组，这种方式使得内存成为内容可寻址的。Linda 通常实现为协作语言——某种普通的编程语言扩展语言，即所谓的计算语言。现在 Linda 已不再用于任何重要的扩展中，但相关联的虚拟共享内存思想被用于 JavaSpaces 中 [FHA99]。

最近开始使用的另一种将消息传递隐藏到虚拟共享内存的技术，是基于划分全局地址空间模型（UPC [UPC]、Titanium [Tita] 和 Co-Array Fortran [C0]）的语言集合。它们是 C、Java 和 Fortran 三种语言分别基于一种虚拟共享内存的显式并行语言（dialect）。与 Linda 或者 OpenMP 等其他共享内存模型不同，划分全局地址空间模型中的共享内存，并包括共享内存到特定处理器的仿射概念。UE 可以读和写彼此的存储器，并进行批量转换。编程模型还考虑了非一致内存访问，使得模型能适用于多种机器，包括 SMP、NUMA 和集群。

252

# OpenMP 简介

OpenMP[OMP] 是一个能够为共享内存计算机创建并行程序的编译制导指令和库函数的集合。OpenMP 结合 C/C++ 或者 Fortran，以创建一种多线程编程语言；也就是说，OpenMP 语言模型基于一个假设：UE 为共享地址空间的线程。

OpenMP 的正式定义包括两个规范，一个针对 Fortran，另一个针对 C 和 C++。虽然它们之间存在细微的差别，但是对于大部分内容，当程序员熟悉基于一种语言的 OpenMP，就可以很轻松地掌握基于另一种语言的 OpenMP。

OpenMP 是基于派生 / 聚合（fork/join）的编程模型。OpenMP 程序开始执行时，只是一个单线程程序；当程序需要并行执行时，就会派生出另外一些线程，形成一个线程组；多个线程并行执行并行区域的代码；在并行区域的最后，线程将等待直到线程组中所有线程都执行到该位置；然后线程聚合在一起。此时，初始或主线程继续执行，直到下一个并行区域（或程序结束）。

OpenMP 创建者的目标就是使得应用程序开发人员能更容易地使用 OpenMP。对于并行程序来说，性能优化是非常重要的。但是这个过程如果会导致编程语言对于软件开发人员变得难以使用，或者难以创建并行程序，又或者程序难以维护，那么这样的优化措施是不可取的。为此，OpenMP 的设计围绕着两个关键原则：*串行等价性*和*递增并行性*。

如果一个程序单线程执行和多线程执行均匀会产生相同结果[⊖]，则称该程序具有*串行等价性*。串行等价程序更易于维护，并且也更容易理解（所以也更容易写）。

*递增并行性*指的将串行程序并行化的一个并行编程类型。程序员从串行程序开始，然后按序在程序中逐次查找值得并行执行的代码片段。这样，并行性是逐步增加的。在并行化每一个代码块后，都进行详细验证，这增大了程序成功并行化的可能性。

并不是始终可以利用递增并行性或创建具有串行等价性的 OpenMP 程序。有时一个并行算法需要完全地重构其串行程序，来获得一个并行程序；有时，程序从最开始构建时就是并行的，没有串行代码。另外，一些并行算法无法通过一个线程来完成，所以不具有串行等价性。但递增并行性和串行等价性仍指导着 OpenMP API 的设计，而且是推荐的实践。

## A.1 核心概念

我们通过讨论一个将字符串输出到标准输出设备的简单程序，来回顾 OpenMP 的核心概念。这个程序的 Fortran 和 C 语言版本如图 A-1 所示。

我们将该程序并行化，即，程序会创建多个线程，每个线程都会输出“E pur si muove”[⊜]。

---

⊖ 由于浮点运算的不可结合性，结果可能略微不同。

⊜ 这是 Giordano Bruno 在 1600 年 2 月 16 日，由于坚持地球绕着太阳转而要被烧死时所说的话。这句拉丁语大概可以翻译为“尽管如此，它还是运动的”。

```
      program simple
      print *,"E pur si muove"
      stop
      end
```

```
#include <stdio.h>
int main()
{
    printf("E pur si muove \n");
}
```

图 A-1 输出字符串到标准输出的 Fortran 和 C 程序

OpenMP 是一种显式并行编程语言，编译器不需推测如何挖掘并行性。程序中的任何并行部分都是由编程人员明确指示出的。为创建 OpenMP 线程，程序员需要指定将要并行执行的代码块。在 C 和 C++ 中通过编译制导（pragma）来完成：

253 ~ 254

```
#pragma omp parallel
```

在 Fortran 中通过如下指令完成：

```
C$OMP PARALLEL
```

现代的程序语言（如 C 和 C++）都是块结构的。但 Fortran 不是。所以，Fortran OpenMP 规范定义了一条指令，来结束并行块：

```
C$OMP END PARALLEL
```

这个模式也应用于 OpenMP 的其他 Fortran 结构中：一条指令用于创建并行结构块，一条在 OMP 之后插入 END 的指令用于结束结构块。

图 A-2 展示了并行程序，每个线程都输出一个字符串到标准输出。

```
      program simple
C$OMP PARALLEL
      print *,"E pur si muove"
C$OMP END PARALLEL
      stop
      end
```

```
#include <stdio.h>
#include "omp.h"
int main()
{
#pragma omp parallel
   {
    printf("E pur si muove\n");
   }
}
```

255

图 A-2 输出一个简单字符串到标准输出的并行程序的 Fortran 和 C 实现

当程序执行时，OpenMP 运行时系统创建一组线程，每个线程都执行并行结构中的指令。如果程序员不特别指出要创建的线程数目，系统就会采用一个默认值。我们稍后会演示如何指定线程数目。在这里，为演示这个例子，假定线程数目为 3。

OpenMP 要求保证 I/O 具有线程级安全性。因此，每个线程输出的输出记录被完全输出，而不受其他线程的干扰。图 A-2 中的程序的输出结果如下所示。

```
E pur si muove
E pur si muove
E pur si muove
```

在这一情况下，每个输出记录都是完全相同的。但是，值得注意的是，尽管每个记录作为一个单元输出，但是记录之间可以以任何方式交叉，所以，程序员不能期望这些记录以确定的顺序输出。

OpenMP 是一个共享内存编程模型，内存模型的细节将在后面讨论。但是大部分实例都会采用的一个规则是：*定义在并行区域之前的变量被线程所共享*。所以图 A-3 中程序的输出结果为：

```
E pur si muove 5
E pur si muove 5
E pur si muove 5
```

```
      program simple
      integer i
      i = 5
C$OMP PARALLEL
      print *,"E pur si muove",i
C$OMP END PARALLEL
      stop
      end
```

```
#include <stdio.h>
#include "omp.h"

int main()
{
  int i=5;

  #pragma omp parallel
  {
    printf("E pur si muove %d\n",i);
  }
}
```

图 A-3 输出一个简单字符串到标准输出的并行程序的 Fortran 和 C 实现

如果一个变量是在并行区域内声明的，就称为线程的*局部变量*或者*私有变量*。在 C 程序中，一个变量的声明可以出现在任意程序块。但是在 Fortran 中，声明只能出现在子程序的开始。

图 A-4 演示了一个包含一个局部变量的 C 程序。这个程序包含一个函数调用：`omp_get_thread_num()`。这个整型函数为 OpenMP 标准运行库（后文会有详细描述）中的函数。对于不同线程，它返回一个用于标识线程的唯一整数值，这个整数值取值范围为 0 到线程数目减一。如果假设线程数量的默认值为 3，则有如下输出（以任意顺序交叉）：

```
c = 0, i = 5
c = 2, i = 5
c = 1, i = 5
```

```
#include <stdio.h>
#include <omp.h>
int main()
{

   int i=5; // a shared variable

#pragma omp parallel
   {
     int c; // a variable local or private to each thread
     c = omp_get_thread_num();
     printf("c = %d, i = %d\n",c,i);
   }

}
```

图 A-4 演示共享变量与局部（或者私有）变量区别的一个简单程序

输出的第一个变量 `c` 是私有变量，每个线程都有一个自己的私有副本，并存储不同值。输出的第二个变量 `i` 是共享的，所以所有线程输出相同的 `i` 值。

在每一个例子中，运行时系统允许选择线程的数量。最常用的方法是，设置环境变量OMP_NUM_THREADS的值，以改变应用于OpenMP程序的系统默认线程数量。例如，在具有csh shell的Linux系统中，为在程序中使用3个线程，在运行程序之前，需执行如下指令：

```
setenv OMP_NUM_THREADS 3
```

这些例子中，应该认为线程数量是一个特定线程数目的请求。特定环境下的系统提供的线程数量可能要小于请求的线程数量。所以，如果一个算法需要知道实际使用的线程数量，必须在并行区域内向系统查询该数目。A.5节会说明如何做到这点。

```
#pragma omp parallel num_threads(3)
```

## A.2 结构块和命令格式

256
~
257

一个OpenMP结构被定义为一个编译制导指令（或pragma）加上一个代码快。但不是任何代码都可以。它必须是结构化的块。也就是说，这个块只有一个顶端入口点，也只有一个底部出口点。

OpenMP程序的结构化块不允许分支进入或分支退出。如果这样做，通常会在编译时产生严重错误。同样地，结构化块不能包含return语句。唯一允许的分支语句是终止整个程序的语句（在Fortran中是STOP，在C中是exit()）。若结构块只包含一个语句，则不需要大括号（在C中）或者end结构指令（在Fortran中）。

如简单示例所示，C或C++的一条OpenMP指令的格式为：

```
#pragma omp directive-name [clause[ clause] ... ]
```

其中，directive-name用于标识该结构块，可选子句[⊖]用于修改结构块。C或C++中的一些OpenMP编译制导例子如下所示：

```
#pragma omp parallel private(ii, jj, kk)
#pragma omp barrier
#pragma omp for reduction(+:result)
```

在Fortran中，情况会复杂一些。我们只考虑简单例子，即固定格式[⊜]的Fortran代码。在这种情况下，一个OpenMP具指令有如下格式：

```
sentinel directive-name [clause[[,]clause] ... ]
```

其中，sentinel可以是：

```
C$OMP
C!OMP
*!OMP
```

其中应用了固定格式源代码行规则。结构中的空格是可选的，行间的连续性是由第6列

258

中的字符所标识。例如，下面三条OpenMP指令是等价的：

```
C$OMP PARALLEL DO PRIVATE(I,J)

*!OMP PARALLEL
*!OMP1 DOPRIVATE(I,J)

C!OMP PARALLEL DO PRIVATE(I,J)
```

⊖ 在本附录中，使用方括号来表示可选的句法元素。

⊜ 固定格式是指在Fortran的旧的版本（Fortran77以及更早的版本）中的语句的固定列约定。

## A.3 工作分摊

在并行结构中，每个线程执行的是相同的代码块。但是，有时我们需要将不同的代码映射到不同的线程上。这称为工作分摊（worksharing）。

OpenMP 中最常用的工作分摊结构是在不同线程间划分循环迭代。在并行编程中，设计关于并行循环的并行算法是非常传统的方法 [X393]。这一方法有时称为循环划分，5.6 节已经做了详细讨论。这种方法中，程序员识别出程序中最耗时的循环。然后，通过将程序的不同循环迭代组映射到不同线程上，以实现并行化。

参考图 A-5 中的程序。在这个程序中，反复调用计算密集型函数 `big_comp()`，以计算中间结果数据，然后把这些数据合并到单个全局结果中。对于这个例子，假设：

- `combine()` 例程运行时间较短；
- `combine()` 函数必须以串行顺序调用。

```
      program loop
      real *8 answer, res
      real *8 big_comp
      integer i, N
      N = 1000
      answer = 0.0;
      do i=1,N
         res = big_comp(i)
         call combine(answer, res)
      end do
      print *,answer
      stop
      end
```

```
#include <stdio.h>
#define N 1000
extern void combine(double, double);
extern double big_comp(int);

int main() {
  int i;
  double answer, res;
  answer = 0.0;
  for (i=0;i<N;i++){
    res = big_comp(i);
    combine(answer,res);
  }
  printf("%f\n", answer);
}
```

图 A-5 一个典型的循环程序的 Fortran 与 C 实现

并行化的第一步是使循环迭代相互独立。完成这一目标的一个可行方法如图 A-6 所示。因为 `combine()` 函数必须以相同顺序调用（和在串行程序中的调用一样），这使并行算法引入了额外的顺序约束。这导致了循环迭代间的依赖关系。我们想并行执行循环迭代，就需要移除这一依赖关系。

```
      program loop
      real *8 answer, res(1000)
      real *8 big_comp
      integer i, N
      N = 1000
      answer = 0.0;
      do i=1,N
        res(i) = big_comp(i)
      end do

      do i=1,N
        call combine(answer,res(i))
      end do

      print *,answer
      stop
      end
```

```
#include <stdio.h>
extern void combine(double, double);
extern double big_comp(int);
#define N 1000

int main() {
  int i;
  double answer, res[N];
  answer = 0.0;
  for(i=0;i<N;i++)
    res[i] = big_comp(i);

  for (i=0;i<N;i++){
    combine(answer,res[i]);
  }
  printf("%f\n", answer);
}
```

图 A-6 一个典型的循环程序的 Fortran 与 C 实现。这个版本中，计算密集型循环被分离和修改，使迭代间相互独立

在这个例子中，假设 combine() 函数是简单的而且不需要花费大量时间来执行。所以，将 combine() 函数移到并行区域外来执行也是可以接受的。我们通过将 big_comp() 计算的每个中间结果存储于一个数组元素中。该数组元素在之后的另外一个循环中按顺序传递给 combine() 函数。此代码转换保留了原始程序的意义（也就是说，并行代码和原始代码的执行结果是一致的）。

通过这一转换，第一个循环的迭代间是相互独立的，可以安全地并行执行。为了将循环迭代分配给多个线程，需要使用 OpenMP 的工作分摊结构。这个结构直接将紧随其后的循环迭代分配给一组线程。之后我们将讨论如何控制循环迭代被调度的方式，但是现在，我们让系统来识别如何将循环迭代映射到线程上。并行版本如图 A-7 所示。

259
~
260

```
      program loop
      real *8 answer, res(1000)
      real *8 big_comp()
      integer i,N
      N = 1000
      answer = 0.0;

C$OMP PARALLEL
C$OMP DO
      do i=1,N
        res(i) = big_comp(i)
      end do
C$OMP END DO
C$OMP END PARALLEL

      do i=1,N
        call combine(answer,res)
      end do

      print *,answer
      stop
      end
```

```
#include <stdio.h>
#include <omp.h>
#define N 1000
extern void combine(double,double);
extern double big_comp(int);

int main() {
  int i;
  double answer, res[N];
  answer = 0.0;

  #pragma omp parallel
  {
    #pragma omp for
    for(i=0;i<N;i++) {
      res[i] = big_comp(i);
    }
  }

  for (i=0;i<N;i++){
    combine(answer,res[i]);
  }
  printf("%f\n", answer);
}
```

图 A-7 一个典型循环程序的 OpenMP 并行化（Fortran 和 C 实现）

OpenMP 的 parallel 结构用于创建一组线程。紧接着是工作分摊结构，在线程间分摊循环迭代：在 Fortran 中用 DO 结构，在 C/C++ 中用 for 结构。因为不存在两个线程更新同一个变量，并且任何不可交换或不可结合的操作（例如，combine() 函数的调用）都以串行顺序执行，所以程序可以正确地并行执行，并保留串行等价性。值得注意的是，根据之前给出的关于变量共享的相关规则，循环控制变量 i 会被线程所共享。OpenMP 规范认为，在并行循环中，一个循环控制索引变量的共享没有任何意义，所以它自动为每个线程创建一个循环控制索引（变量 i）的副本。

默认情况下，在所有工作分摊结构的后面都会有隐含的 barrier 操作。也就是说，所有线程到达结构的末尾后均等待，只有等到所有线程都到达时，线程才继续执行。可以通过在工作分摊结构中添加 nowait 子句消除这种 barrier：

```
#pragma omp for nowait
```

261

但要慎用 nowait 子句，因为在大多数情形中，都需要这个 barrier 来阻止竞态条件。

在 OpenMP 中，存在其他两种常用的工作分摊结构：single 结构和 sections 结构。single 结构定义一个代码块，该代码块将被第一个利用该结构的线程所执行。其他线程跳过该结构，并在 single 结构的末端等待隐含的 barrier 操作（除非使用了 nowait 子句）。我们在第 6 章的 OpenMP 归约示例中使用了这种结构。当我们将在并行区域中创建的线程数目复制到一个共享变量中时，为了确保仅有一个线程对这个共享变量进行写操作，需要将这条语句放到一个 single 结构中：

```
#pragma omp single
    num_threads = omp_get_num_threads();
```

sections 结构用于构建一个程序区域。在这个区域中，把不同的代码块分配给不同的线程。每块代码均定义成一个 section 结构。在示例中，没有使用 sections 结构，因此这里不再详述。

在 OpenMP 程序中，在一个并行结构后面紧跟着一个工作分摊结构是非常常见的情况。例如，在图 A-7 中，有如下代码：

```
#pragma omp parallel
{
  #pragma omp for
    ...
}
```

为了简化代码，OpenMP 定义了一种组合结构。

```
#pragma omp parallel for
    ...
```

这与 parallel 结构和 for 结构独立放置是一致的。

## A.4 数据环境子句

OpenMP 是一种相对简单的 API。在使用 OpenMP 时，大多数困难源于如何在线程间共享数据以及如何实现数据初始化与 OpenMP 之间的交互。本附录中，我们将解决某些更常见的问题，如果需要完全理解 OpenMP 相关的数据环境，可以阅读 OpenMP 规范 [OMP]。内存同步的细节在 6.3 节中有讨论。 262

我们首先定义在描述 OpenMP 数据环境时所用到的术语。在一个程序中，变量是一个容器，该容器拥有一个名称和一个值（更具体地说，是内存或寄存器中的一个存储位置）。当程序运行时，变量可以读写（与仅能够读取的常量相对）。

在 OpenMP 中，绑定到一个给定名称的变量依赖于该名称是出现在并行区域的前面，还是出现在并行区域内，抑或是出现在并行区域之后。当变量的声明在并行区域之前时，它默认是共享的，并且该名称始终绑定相同的变量。

但是，OpenMP 包含一些可以添加到 parallel 结构和工作分摊结构中以控制数据环境的子句。这些子句影响该名称对应的变量。private(list) 子句指示编译器为每个线程创建列表中所包含变量的一份私有（或局部）副本。private 列表中的变量必须已经定义，并且绑定在并行区域之前声明的共享变量。这些新的私有变量的初始值是未定义的，因此它们需显式初始化。另外，在并行区域后，出现在 private 子句中的名称所绑定的变量的值是也未定义的。

例如，5.6 节给出了一个简单地利用梯形规则进行积分的程序，该程序由一个简单的主循环组成。在循环中，在 x 值的变化范围内，计算相应被积函数的值。x 变量是一个临时变量集，并且用于循环的每个迭代中。因此，通过给予每个线程这个变量的一个副本，就可以消除该变量所隐含的依赖性。利用 private(x) 子句就可以完成这个任务，如图 A-8 所示（后文将讨论这个示例中的 reduce 子句）。

```
#include <stdio.h>
#include <math.h>
#include <omp.h>

int main () {
   int i;
   int num_steps = 1000000;
   double x, pi, step, sum = 0.0;

   step = 1.0/(double) num_steps;

#pragma omp parallel for private(x) reduction(+:sum)
   for (i=0;i< num_steps; i++)
   {
      x = (i+0.5)*step;
      sum += 4.0/(1.0+x*x);
   }
   pi = step * sum;
   printf("pi %lf\n",pi);
   return 0;
}
```

图 A-8 利用梯形规则进行积分的（等价于 $\int_0^1 \frac{4}{1+x^2}\mathrm{d}x$）的 C 程序

263 其他一些改变变量在线程间共享方式的最常用子句包括以下几个。

- firstprivate(list)。和 private 一样，出现在列表中的每个名字，都会在每个线程中创建一个具有该名字的新的私有变量。但是，和 private 不同的是，这些新创建的变量会初始化，初始化的值为该变量在包含该 firstprivate 子句的结构之前的代码中所具有的值。这个子句在 parallel 结构和工作分摊结构中都可以使用。
- lastprivate(list)。同样，list 中的每个名字在每个线程均中创建相应的局部（或者私有）变量。但在这种情况下，最后一个循环迭代中的私有变量的值将被复制到出现在 OpenMP 结构（包含该 lastprivate 子句）之后的代码中，该名称对应的变量上。

变量通常只可以出现在一个数据子句的列表中。firstprivate 和 lastprivate 除外，因为在很多问题中，很可能一个私有变量既需要一个已定义的初始化值，又需要将一个数据传递到 OpenMP 结构后面的代码中。

在图 A-9 中提供了一个使用这些子句的示例。我们为每个线程声明了 4 个私有变量，并对其中三个变量进行值赋：h=1，j=2 和 k=0。在并行循环之后，输出了 h、j 和 k 的值。其中，只有变量 k 是明确定义的。对于每个线程，执行一次循环迭代，变量 k 就增 1。因此，它记录了每个线程处理的迭代数量。

```
#include <stdio.h>
#include <omp.h>
#define N 1000

int main()
{
  int h, i, j, k;
  h = 1; j = 2; k = 0;
#pragma omp parallel for private(h) firstprivate(j,k) \
                         lastprivate(j,k)
  for(i=0;i<N;i++) {
        k++;
        j = h + i; //ERROR: h, and therefore j, is undefined
  }

  printf("h = %d, j = %d, k = %d\n",h, j, k); //ERROR j and h
                                              //are undefined
}
```

图 A-9 演示 private、firstprivate、lastprivate 子句的使用方法的 C 程序。这个程序在 printf 函数被调用时会出现错误，因为变量 h 和 j 都是未定义的。反斜杠用来将编译指令延续到下行 264

因为 lastprivate 子句，被传递到循环之外的值是执行最后一个循环迭代（即 i=999 的循环迭代）的线程的 k 的值。h 和 j 的值是未定义的，但它们未定义的原因有所不同。变量 j 未定义的原因是，在并行循环中，它被赋值为一个未初始化的变量（h）与一个变量的累加和。变量 h 的问题则更复杂些。它被声明为一个 private 类型的变量，但在循环内部它的值未改变。但是，OpenMP 规定，如果一个名字出现在 private 子句中，则该 OpenMP 结构之后的代码区域中与该名字绑定的变量都未定义。因此，parallel for 之后的输出语句中，h 没有一个已定义的值。

还有一些其他影响变量共享方式的子句，但在本书中没有使用它们，因此不再讨论它们。

最后，我们要讨论的影响数据共享方式的子句是 reduction 子句。第 6 章已经讨论了归约。归约是一通过一个二元、可结合的运算符，将一个数据集合并成为单个值的操作。归约非常常用，大多数并行编程环境中都有归约实现。在 OpenMP 中，归约子句定义一个变量名列表和一个二元运算符。对于列表中的每个名字，创建一个私有变量，并初始化为该二元运算符的单位元素值（例如，加法运算的单位元素值为 0）。每一个线程对与列表中名称相应的局部变量副本进行归约操作。在包含 reduction 子句的结构末尾，把这些局部值合并到该 OpenMP 结构之前的相应变量中，以产生单个值。把这个值赋给包含归约操作的 OpenMP 结构之后的代码区域中的具有相同名字的变量。

图 A-8 中给出了一个归约示例。在这个示例中，reduction 子句被应用于“+”运算符，以计算数据总和，并将结果保留在变量 sum 中。

尽管最常用的归约是求和，但 OpenMP 还是在 Fortran 中为 +、*、-、.AND.、.OR.、.EQV.、.NEQV.、MAX、MIN、IAND、IOR 和 IEOR 提供了归约操作的支持。在 C 和 C++，OpenMP 支持标准 C/C++ 运算符 *、-、&、|、^、&& 以及 || 的归约操作。C 和 C++ 在语言定义中不包括大量有意义的固有函数，例如“min”或“max”。因此，OpenMP 没有相应的归约操作；如果需要它们，程序员必须自己来编写对应代码。在 OpenMP 规范 [OMP] 中有关于 OpenMP 归约更详细的描述。

## A.5 OpenMP 运行时库

应该尽可能利用编译制导指令来表示 OpenMP 语法。然而，某些功能只能利用运行时库函数来实现。在运行时库中，最常用的一些函数如下所示。

- `omp_set_num_threads()`，该函数具有一个整型参数，操作系统会根据这一数值在
265 随后的并行区域中创建相应数目的线程。
- `omp_get_num_threads()`，该整型函数返回当前线程组中的实际线程数目。
- `omp_get_thread_num()`，该整型函数返回线程的 ID，线程 ID 的取值范围为 0 到线程的总数减 1。ID 为 0 的线程是主线程。
- 锁函数进行锁的创建、使用和销毁。这些将在后面的同步结构中描述。

在图 A-10 中，给出了关于这些函数使用方式的一个简单示例。该程序在标准输出上输出了线程的 ID 和线程的数目。

```
#include <stdio.h>
#include <omp.h>

int main() {

    int id, numb;

    omp_set_num_threads(3);

 #pragma omp parallel private (id, numb)
   {
    id = omp_get_thread_num();
    numb = omp_get_num_threads();
    printf(" I am thread %d out of %d \n",id,numb);
   }
}
```

图 A-10 常用运行时库函数使用的 C 程序

图 A-10 中程序的输出将类似于下面的形式：

```
I am thread 2 out of 3
I am thread 0 out of 3
I am thread 1 out of 3
```

输出的记录将是完整的并且非重叠的，因为 OpenMP 要求 I/O 库是线程级安全的。但是，哪一个线程在什么时间输出它的记录没有指定，因此输出记录的任何有效交错都有可能发生。

## A.6 同步

许多 OpenMP 程序仅使用 `parallel` 和 `parallel for`（在 Fortran 中是 `parallel do`）结构来编写就足够了。但是，存在一些算法需要对变量共享方式进行更小心地控制。当多个线程都要读和写共享数据时，程序员必须确保这些线程之间不相互干扰，以使得线程无
266 论怎样调度，程序都返回相同的结果。对于一个多线程程序，这是至关重要的，任何语义上允许的指令干扰都有可能发生。因此，程序员必须管理对共享变量的读和写，以确保线程读取正确的数值，多个线程不在同一时间写同一个变量。

同步是管理共享资源的方法，用于实现线程无论怎样调度，读和写都可以以正确的顺序

发生。6.3 节对同步的概念进行了详细讨论。在此将主要介绍 OpenMP 中同步的语法和使用方式。

回顾本章节前面图 A-5 中的基于循环的程序。我们使用了该示例来介绍 OpenMP 中的工作分摊法，假设结果（res）组合的计算不占用太多的时间，并且必须以串行顺序发生。因此，可以将中间结果存储在一个共享数组中并在后面（一个串行区域中）将结果组合为最终的答案。

但是，在通常情况下，只要累加不相互影响，big_comp() 的结果可以以任意顺序累加。为了使得问题更有趣，假设 combine() 例程和 big_comp() 例程都是耗时的，并且执行时间是不可预测的且变化范围较大。因此，我们需要将 combine() 函数放置到并行区域中，并且使用同步结构以确保对 combine() 函数的并行调用不相互干扰。

OpenMP 中主要的同步结构如下所示。

- flush，它定义一个同步点，在该点，内存一致性强制执行。这是比较复杂的。现代计算机基本上都会把数值存放在寄存器或缓冲区中，但不能保证在任何时刻都与计算机内存中的内容一致。有些内存一致性协议保证所有处理器最终看到的是单个地址空间，但不保证内存引用在任意时刻都能及时更新并保持一致。flush 的语法如下所示。

```
#pragma omp flush [(list)]
```

- 其中，list 是一个由分号隔离的需要刷新的变量列表。如果该列表省略，则所有对调用线程可见的变量都将刷新。程序员很少需要显式地调用 flush，因为在大多数需要它的地方，它会自动地插入。通常，程序员只需要构建他们的低级同步原语就够了。
- critical，为互斥现象创建一个临界区。也就是说，同一时刻只能有一个线程能执行临界区中的结构块代码。其他线程将在该结构的顶部等待。临界区的语法如下 267
  所示。

```
#pragma omp critical [(name)]
{ a structured block }
```

- 其中，name 是一个标识符，用于支持临界区间的集合不相交性。临界区在其入口和出口处隐含了 flush 调用。
- barrier，提供一个同步点，线程到达该点后将等待，直到线程组中的每一个成员都到达，线程才会继续执行。barrier 的语法如下所示。

```
#pragma omp barrier
```

可以显式地添加 barrier，但是它也会在需要它的地方隐式地调用（例如，在并行结构或者工作分摊结构的末尾）。barrier 中隐含了一个 flush。

第 6 章进一步讨论了临界区、barrier 和 flush。

回到图 A-5 中的示例，如果增加一个互斥操作，就能够保证在并行循环内部安全地执行 combine() 例程的调用。我们将采用 critical 结构，如图 A-11 所示。需要注意的是，我们必须为每个线程创建变量 res 的一个私有副本，以避免循环迭代间的访问冲突。

```
#include <stdio.h>
#include <omp.h>
#define N 1000
extern void combine(double,double);
extern double big_comp(int);

int main() {
   int i;
   double answer, res;
   answer = 0.0;
#pragma omp parallel for private (res)
   for (i=0;i<N;i++){
      res = big_comp(i);
#pragma omp critical
      combine(answer,res);
   }
   printf("%f\n", answer);
}
```

图 A-11 图 A-5 中的程序的并行版本。但是，在这种情形中，假设可以以任意顺序调用 combine() 函数，只要保证同一时刻只有一个线程运行这个函数就可以。程序中利用了 critical 结构来保证这一需求

268

对于大多数程序员来说，OpenMP 中的高层同步结构已经够用了。但是，也存在高层同步结构无法使用的情况。两种常见的情况如下所示。

- 问题中要求的同步协议不能通过 OpenMP 的高层同步结构所表达。
- OpenMP 的高层同步结构所产生的并行开销太大。

为了解决这些问题，OpenMP 运行时库包含一些提供锁功能的低级同步函数。锁函数的原型定义在 omp.h 文件中。“锁”使用了一种不透明的数据类型 omp_lock_t，omp_lock_t 同样定义在 omp.h 文件中。一些关键的函数如下所示。

- void omp_init_lock(omp_lock_t *lock)，用于初始化锁。
- void omp_destroy_lock(omp_lock_t *lock)，用于销毁锁，并释放与该锁相关的所有内存。
- void omp_set_lock(omp_lock_t *lock)，用于设置或获取锁。如果锁是空闲的，线程调用 omp_set_lock() 后，将获得锁，并继续执行。如果锁被另外一个线程所拥有，线程调用 omp_set_lock() 后，将等待，直到锁可用为止。
- void omp_unset_lock(omp_lock_t *lock)，用于解除或释放锁，以便其他线程能够获取它。
- void omp_test_lock(omp_lock_t *lock)，用于测试或者查询锁是否可用。如果锁可用，则程序获取锁并继续执行。锁的测试和获取是自动完成的。相反，如果锁不可用，该函数将返回 false(nonzero)，并且主调线程继续执行。这个函数使得线程在等待锁的同时还能够做一些有价值的工作。

锁函数保证了锁变量本身在线程间一致地更新，但并不实现对其他变量的隐含的 flush 操作。因此，使用锁的程序员必须按需显式地调用 flush。图 A-12 是一个使用 OpenMP 锁的示例程序。该程序在程序的开始部位声明并初始化两个锁变量。因为在并行区域之前声明，所以锁变量被所有的线程所共享。在并行区域内，第一个锁用于确保在同一时刻仅有一个线程向标准输出上输出消息。该锁用于确保每个线程中的两个 printf 语句连续执行，不被其

他线程中的 printf 语句穿插干扰。第二个锁用于确保在同一时刻仅有一个线程执行 go_for_it() 函数，通过使用 omp_test_ lock() 函数，实现线程在等待锁的同时能够做一些有价值的工作。并行区域执行完毕后，通过调用 omp_lock _destroy() 函数来释放与锁相关的内存。 269

```
#include <stdio.h>
#include <omp.h>

extern void do_something_else(int); extern void go_for_it(int);

int main() {
   omp_lock_t lck1, lck2; int id;

   omp_init_lock(&lck1);
   omp_init_lock(&lck2);

#pragma omp parallel shared(lck1, lck2) private(id)
   {
      id = omp_get_thread_num();

      omp_set_lock(&lck1);
      printf("thread %d has the lock \n", id);
      printf("thread %d ready to release the lock \n", id);
      omp_unset_lock(&lck1);

      while (! omp_test_lock(&lck2)) {

        do_something_else(id); // do something useful while waiting
                               // for the lock
      }
      go_for_it(id); // Thread has the lock

      omp_unset_lock(&lck2);
    }
    omp_destroy_lock(&lck1);
    omp_destroy_lock(&lck2);
}
```

图 A-12 OpenMP 中锁函数使用方式的代码示例

## A.7 schedule 子句

基于循环的并行算法的性能关键在于将循环的迭代合理地分配到所有线程，使得线程间达到负载平衡。OpenMP 编译器试图为程序员自动化地实现这一点。尽管编译器擅长管理数据依赖性，并能进行高效的通用优化，但是它们不善于理解特定算法的访存模式和不同循环迭代的执行时间的差别。为了获得最佳性能，程序员需要告诉编译器如何将循环迭代划分给每个线程。

可以通过在 for 或 do 工作分摊结构中添加一个 schedule 子句来实现。在 C 和 Fortran 中，schedule 子句形式如下。

```
schedule( sched [,chunk])
```

270

其中，sched 可以是 static、dynamic、guided 或者 runtime 中任意一个，chunk 是一个可选的整型参数。

- schedule(static [,chunk])，循环迭代空间被划分为大小为 chunk 的块。如

果 chunk 省略，则块大小是使得每个线程拥有近似相等大小的块的对应数值。块以轮询方式被分发给线程组。例如，当线程数为 3，块的大小为 2 时，12 个循环迭代将创建为 6 个块，它们包含的循环迭代分别是 (0，1)、(2，3)、(4，5)、(6，7)、(8，9) 和 (10，11)，块分配给线程的方式是：[(0，1)，(6，7)] 分配给 0 号线程，[(2，3)，(8，9)] 分配给 1 号线程，[(4，5)，(10，11)] 分配给 2 号线程。

- schedule(dynamic [,chunk])，循环迭代空间被划分为大小为 chunk 的块。如果 chunk 省略，则块的大小为 1。最初给每个线程都分配一个循环迭代块来处理。剩余的块放置在一个队列中。当处理完当前块之后，线程就从队列中取出接下来需要处理的循环迭代块。这样持续下去，直到所有的循环迭代块处理完。
- schedule(guided [,chunk])，是对动态调度的一种优化版本，以降低调度开销。与动态调度一样，循环迭代被划分为一些块，初始状态下，给每个线程分配一个迭代块，当完成分配的块后，线程就再接收一个块来处理。区别在于块的大小方面：第一块的大小是与实现相关的，但仍相对较大；后续块的大小迅速下降，最终降到 chunk 所指定的值。这种方法拥有动态调度的优点，由于起始块较大，运行时的调度决策数目相对减小，因此并行开销极大降低。
- schedule(runtime)，runtime 调度规定，实际的调度方法和循环迭代块的大小取决于环境变量 OMP_SCHEDULED 的值。这使得一个可执行程序可以实现不同的调度方法，而不用在每次调度试验中重新编译代码。

参照图 A-13 中基于循环的程序。因为 schedule 子句在 Fortran 和 C 语言中实现是相同的，所以我们只讨论在 C 中的情况。如果当程序运行时，不同循环迭代的运行时间不相同，而且是不可预测，则静态调度法可能不够高效。这时，我们会使用动态调度法。然而，调度开销是一个严重的问题，因此为了使调度决策的数目最小，我们将每个调度块的大小设置为 10 个循环迭代。然而，并不存在明确的规律来确定最优的调度方法和块大小，OpenMP 程序员通常需要试验各种调度方法和块的大小，直到获得最优值为止。例如，图 A-13 中的并行循环程序中，如果迭代块设置得足够小，能使得迭代工作均匀地分布于线程间，则静态调度方
271 法同样也可以取得很高的效率。

```
#include <stdio.h>
#include <omp.h>
#define N 1000
extern void combine(double,double);
extern double big_comp(int);

int main() {
   int i;
   double answer, res;
   answer = 0.0;
#pragma omp parallel for private(res) schedule(dynamic,10)
  for (i=0;i<N;i++){
     res = big_comp(i);
#pragma omp critical
     combine(answer,res);
  }
  printf("%f\n", answer);
}
```

图 A-13 图 A-11 中程序的并行版本，并做了一些更改，以演示 schedule 子句的使用

## A.8 程序语言的其余部分

本附录仅讨论了本书中所涉及的 OpenMP 特征。尽管已经涵盖了 OpenMP 大部分最常用的结构，但是对使用 OpenMP 感兴趣的程序员还应完整地阅读 OpenMP 规范 [OMP]。所有 API 的定义仅有 50 多页。OpenMP 规范还包括超过 25 页的程序示例。[CDK+00] 是介绍 OpenMP 语法的书籍，[Mat03] 中提供了对 OpenMP 的抽象的描述和一些关于未来方向的想法。OpenMP 的相关文献正在逐渐地增加；各种 OpenMP 研讨会 [VJKT00、EV01、EWO0l] 的会议中也提出了大量的 OpenMP 应用。 272

# MPI 简介

MPI（Message Passing Interface）是分布式内存并行计算机的标准编程环境。MPI 的核心构造是消息传递：一个进程[⊖]将信息打包成为一条消息然后把这条消息发送到目标进程。但 MPI 远远不仅仅是简单消息传递，MPI 还包含有很多用于同步进程、将分布在一系列进程中的数字相加，将数据分发到一系列进程中等其他操作的例程。

为了提供一个能够在集群、MPP 甚至在共享内存计算机运行的消息传递环境，MPI 在 20 世纪 90 年代早期提出。MPI 以库的形式发布，官方规范定义为 C 和 Fortran 的绑定（尽管其他语言的绑定也有定义）。现在绝大多数 MPI 程序员使用的是 1.1 版本的 MPI（于 1995 年发布）。升级后的规范 MPI 2.0 于 1997 年发布，其中包含并行 I/O、动态进程管理、单边通信和其他的新特性。但是由于它对原有的规范增加了复杂的内容，所以即便 6 年之后，只有屈指可数的 MPI 实现支持 MPI 2.0。因此，这里讨论的是 MPI 1.1.

现在有几种常见的 MPI 实现。最常见的是 LAM/MPI [LAM] 和 MPICH [MPI]. 这两者都可以免费下载并通过它们附带的说明简单地安装。它们支持非常多种类的并行计算机，像 Linux 集群、NUMA 计算机和 SMP。

## B.1 概念

传递消息的基本概念看上去很简单：一个进程发送一条消息，另一个进程接收这条消息。
273 但是挖掘得更深一些，消息传递背后的细节会非常复杂：如何在系统中缓冲消息？一个进程在接收或发送消息时还能做其他有用的工作吗？怎样使得一条消息被标识，这样它发送后始终可以匹配到目的接收者？

MPI 的成功就是因为它很好地解决了这些（和其他的）问题。方法是基于两个 MPI 的核心要素：*进程组*和*通信上下文*。一个进程组就是计算中的进程集合。在 MPI 中，当程序启动的时候，所有参与一个计算的进程同时启动，它们属于一组。但是随着计算的进行，程序员能够将这些进程分成更小的子组，然后精细地控制这些组的交互。

*通信上下文*为组织相关通信集提供了一种机制。在任何消息传递系统中，消息必须是标记的，这样消息就能够送到它们的目的地上。在 MPI 中消息标签有发送方 ID、接收方 ID 和一个整数标签组成。receive 语句包括了指示源和标签的参数，其中两个参数中的每一个或者两个参数都可能是通配符。所以，在进程 *i* 中执行一条 receive 语句的结果是传递一个目的是进程 *i* 且类型和源都与 receive 语句相匹配的消息。

虽然简单，但是标识一条消息只靠源、目的和标签在复杂的应用程序中是不够的，特别是那些包含库或者从其他程序中重用功能的程序。通常情况下，应用程序程序员不知道这些代码中的细节，如果在这些库中调用了 MPI，那么就可能出现应用程序中有些消息和库函数

⊖ 在 MPI 中 UE 是进程。

中的消息标签、目标 ID、源 ID 恰好相同的情况。这可能会导致在库消息被传递到应用程序代码或者应用程序代码传递消息到库的时候程序出错。处理这种问题的一种方法是让库的作者指定用户必须避免使用的保留标签，已证明这种方法是笨拙的、容易出错的，因为它要求编码者仔细阅读和遵循文档中的说明。

MPI 的解决方案是基于通信上下文[⊖]的概念。每一个发送（和它的最终消息）和接收都属于一个通信上下文，而且只有那些共享一个通信上下文的通信事件可以匹配。因此即便有些消息共享一个源、一个目标、和一个标签，只要它们有不同的上下文，它们不会和其他消息混淆。消息上下文会动态生成并且保证它的独特性。

在 MPI 中，进程组和通信上下文组合到一个叫做通信域的对象中。除了一些非常少的特例外，MPI 中的函数包括通信域的一个引用。当程序启动时，运行时系统会创建一个叫做 MPI_COMM_WORLD 的公用通信域。 274

在大多数情况下，MPI 程序员仅仅需要一个通信域，也就是 MPI_COMM_WORLD。创建和操作一个通信域是简单的，只有那些写可重用软件组件的程序员需要去做这些事。因此，管理通信域不在本书的讨论范围内。

## B.2 开始

因为 MPI 标准没有规定一个工作怎样开始，所以不同的 MPI 在不同情形下的实现之间有着巨大的差别（比如，作业是以交互方式启动还是由批调度程序启动）。对于使用 MPI 1.1 以交互方式启动作业，最常用的方法是在获取配置文件中列表的节点处启动所有与 MPI 相关的进程。所有进程执行相同的程序。完成这个任务的命令通常叫做 mpirun，其使用程序名作为参数。遗憾的是，MPI 标准没有定义 mpirun 的接口，因此它会因为 MPI 的实现不同而有差异。这些细节可以在每个实现的文档中找到 [LAM.MPI]。

所有 MPI 程序都有一些基本元素。考虑图 B-1 中的程序。

```
#include <stdio.h>
int main(int argc, char **argv) {
    printf("\n Never miss a good chance to shut up \n");
}
```

图 B-1 将简单字符串标准输出到标准输出的程序

我们在将这个程序修改为多进程执行的并行程序的过程中将探讨 MPI 所必要的因素。首先，MPI 的函数原型需要定义。这个工作在 MPI 的头文件中完成。

```
#include <mpi.h>
```

接着，MPI 环境必须初始化。作为初始化的一部分，创建被全部进程共享的通信域，也就是，先前提到的 MPI_COMM_WORLD:

```
MPI_Init(&argc, &argv);
```

命令行参数被传递到 MPI_Init，这样 MPI 环境就能够通过添加程序自己的命令行参数 275

⊖ Zipcode [SSD+94] 是唯一早于 MPI 且有独立的通信上下文的消息传递库。

（对于程序员是透明的）影响程序的行为。这个函数返回一个用于表明函数调用成功或者失败的整型状态标志。除了少数以外，所有 MPI 函数都返回这个标志。这个标志可能取的值在 MPI 头文件（mpi.h）中描述。

虽然没有要求，但是几乎每一个 MPI 程序都使用进程组中的进程数量和各个进程在组中的序号[⊖]来指导计算，如 5.4 节所述。这些信息可以通过调用 MPI_Comm_size 和 MPI_Comm_rank 函数获得。

```
int num_procs; // the number of processes in the group
int ID;// a unique identifier ranging from 0 to (num_procs-1)

MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
MPI_Comm_rank(MPI_COMM_WORLD, &ID);
```

当 MPI 程序结束运行时，整个环境需要显式地关闭。这个将会由下面的函数完成：

```
MPI_Finalize();
```

在很简单的程序中使用这些元素，我们得到了并行程序（见图 B-2）。

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char **argv) {
    int num_procs; // the number of processes in the group
    int ID; // a unique identifier ranging from 0 to (num_procs-1)

    if (MPI_Init(&argc, &argv) != MPI_SUCCESS) {
      // print error message and exit
    }

    MPI_Comm_size (MPI_COMM_WORLD, &num_procs);
    MPI_Comm_rank (MPI_COMM_WORLD, &ID);
    printf("\n Never miss a good chance to shut up %d \n",ID);

    MPI_Finalize();
}
```

276

图 B-2 每个进程都标准输出简单字符串的并行程序

如果在程序中的任意一点检测到一个致命错误，程序可能会有关闭。如果这一点没有谨慎地完成，那么进程可能会出现停留在一些节点的情况。这些进程称作*孤进程*，原则上，它们可能会“永恒”地等待着与组内的其他进程交互。下面的函数告诉 MPI 运行时环境去尽可能地尝试关闭 MPI 程序中的所有进程：

```
MPI_Abort();
```

## B.3 基本点对点消息传递

MPI 中的点对点消息传递例程是从一个进程传递消息到另一个进程。在 MPI 1.1 针对点对点通信有超过 21 个函数。大量的点对点通信函数不仅提供了优化通信缓冲区所需要的控制手段，还规范了怎样通信及计算的重叠。

⊖ 序号是指范围从 0 到进程数量减一的整数，它用来表明每个进程在进程组中的位置。

在 MPI 1.1 中，最常用的消息传递函数是在图 B-3 中定义的阻塞发送 / 接收函数。MPI_Send() 函数在将缓冲区（buff）传递给系统同时能够安全重用的时候返回。在接收方，MPI_Recv() 函数在当缓冲区（buff）接收到消息并准备使用的时候返回。

```
int MPI_Send (buff, count, MPI_type, dest, tag, Comm);

int MPI_Recv (buff, count, MPI_type, source, tag, Comm, &stat);


buff                指向缓冲区的指针，其类型与 MPI_type 相兼容
int count           buff 中指定类型的项数
Mpi_type            buff 中项的类型，最常用的类型是 MPI_DOUBLE、
                    MPI_INT、MPI_LONG 和 MPI_FLOAT
int source          发送消息的进程序号。常数 MPI_ANY_SOURCE 能被
                    MPI_Recv 使用以接收任意源的消息
int dest            接收消息的进程的序号
int tag             通信中一个用于标识消息的整数。常数 MPI_ANY_TAG
                    是可以匹配所有标记值的通配符
MPI_Status stat     在接收消息时一个保存消息状态的结构
MPI_ COMM Comm      MPI 的通信域，通常为默认的 MPI_COMM_WORLD
```

图 B-3　C 语言下 MPI 1.1 的标准阻塞点对点通信例程

MPI_Status 数据类型是在 mpi.h 中定义的。这个状态变量用来刻画一条接收到的消息。如果 MPI_ANY_TAG 在 MPI_Recv() 中用到，例如对于这条消息真实的标记将从状态变量中抽取为 status.MPI_TAG。

在图 B-4 中的程序提供了一个如何使用基本的点对点消息传递函数的样例。在这个程序中，一条消息往返于两个进程。这个程序（也称作“乒乓球”程序）经常被用做并行计算机中通信系统性能的基准。

```
#include <stdio.h> // standard I/O include file
#include "memory.h" // standard include file with function
                    // prototypes for memory management
#include "mpi.h" // MPI include file

int main(int argc, char **argv) {
  int Tag1 = 1; int Tag2 = 2; // message tags
  int num_procs; // the number of processes in the group
  int ID; // a unique identifier ranging from 0 to (num_procs-1)
  int buffer_count = 100; // number of items in message to bounce
  long *buffer; // buffer to bounce between processes
  int i;
  MPI_Status stat; // MPI status parameter

  MPI_Init(&argc,&argv); // Initialize MPI environment
  MPI_Comm_rank(MPI_COMM_WORLD, &ID); // Find Rank of this
                                      // process in the group
```

图 B-4　C 语言下使用堵塞点对点通信函数“连接”两个进程的 MPI 1.1 程序

```
    MPI_Comm_size (MPI_COMM_WORLD, &num_procs); // Find number of
                                                // processes in the group
    if (num_procs != 2) MPI_Abort(MPI_COMM_WORLD, 1);

    buffer = (long *)malloc(buffer_count* sizeof(long));

    for (i=0; i<buffer_count; i++) // fill buffer with some values
       buffer[i] = (long) i;

    if (ID == 0) {
       MPI_Send (buffer, buffer_count, MPI_LONG, 1, Tag1,
                 MPI_COMM_WORLD);
       MPI_Recv (buffer, buffer_count, MPI_LONG, 1,
                 Tag2, MPI_COMM_WORLD, &stat);
    }
    else {
       MPI_Recv (buffer, buffer_count, MPI_LONG, 0, Tag1,
                 MPI_COMM_WORLD,&stat);
       MPI_Send (buffer, buffer_count, MPI_LONG, 0, Tag2,
                 MPI_COMM_WORLD);
    }

    MPI_Finalize();
  }
```

图 B-4 （续）

这个程序以常见的头文件和声明开始。在程序中，初始化 MPI 作为一开始的可执行语句。其实这个工作是可以在后面做的，关于初始化语句的唯一硬性要求就是初始化必须在调用 MPI 例程前发生。接着，我们决定每个进程的序号和进程组的大小。为了让整个程序短小精悍，我们假设在进程组中只有两个进程。接着为缓冲区分配内存。

整个通信将会自动变成两部分。在序号为 0 的进程中，我们发送然后接收消息，在其他进程中，我们接收然后发送消息。用这种方法（一个进程发送然后接收消息，而另一个进程接收然后发送消息）匹配对阻塞发送消息机制和阻塞大量消息是非常重要的。为了彻底理解这一点，考虑以下问题：有两个进程（ID 0 和 ID 1）都需要发送消息与接收消息。因为这是一个 SPMD 程序，所以极有可能把它写成如下形式：

```
int neigh; // the rank of the neighbor process

If (ID == 0) neigh = 1; else neigh = 0;

MPI_Send (outgoing, buffer_count, MPI_LONG, neigh, Tag1,
          MPI_COMM_WORLD);

// ERROR: possibly deadlocking program
MPI_Recv (incoming, buffer_count, MPI_LONG, neigh, Tag2,
          MPI_COMM_WORLD, &stat);
```

这段代码非常简单、紧凑，但是在有些情况下（比如，消息占的空间非常大），系统可能没办法找到足够的空间去发送消息，直到它们在通信结束的时候被复制到目标进程的缓冲区里。因为消息发送将会阻塞，直到系统缓冲区可以重用，接收函数将永远不会被调用，这样就会导致程序死锁。因此，为了写前面的代码，最安全的方法就是像我们在图 B-4 中做的那样，把通信分开。

## B.4 集合操作

除了点对点消息传递例程之外，MPI 也包含组内所有进程一同工作以实现复杂通信的集合操作。这些集合通信操作对于 MPI 编程来说是非常重要的，事实上，很多 MPI 程序主要包括集合操作，而完全没有成对的消息传递。

最常用的集合操作包含如下函数。

- MPI_Barrier。定义了一个同步点，所有进程都必须到达这个同步点才能继续运行。对于 MPI 来说，这就意味着每个使用该通信域的进程在它们继续执行之前必须调用该函数。这个函数的细节见第 6 章。
- MPI_Bcast。将来自一个进程的消息广播到进程组中的所有进程。
- MPI_Reduce。归约操作获得分散在进程组中的一个值集，这些值位于 inbuff 指向的缓冲区中，然后用指定的二元操作符计算它们。为了让这个函数意义，问题中的操作必须符合结合律。对于最常见二元函数的例子是加法和在一组数中找出最大或最小值。注意，仅仅可以在所指定的目的进程中得到最终的归约值（位于 outbuff 指向的缓冲区）。如果这个值要被所有进程所用，我们需要用到这个函数的变体 MPI_ALL_reduce()。归约的细节见第 6 章。

277 ~ 279

这些函数的语法在图 B-5 中定义。

```
int MPI_Barrier (COMM);

int MPI_Bcast (outbuff, count, MPI_type, source, COMM);

int MPI_Reduce (inbuff, outbuff, count, MPI_type, OP, dest, COMM);

inbuff、outbuff    指向与 MPI_type 相兼容缓冲区的指针
int count          buff 中包含指定类型的条目个数
MPI_type           定义在 mpi.h 中缓冲区里条目的类型
int source         发送消息进程的序号
int dest           接收最后归约输出结果的进程的序号
MPI_COMM Comm      MPI 通信域，通常使用默认的 MPI_COMM_WORLD
OP                 用于归约的运算，定义在 mpi.h 中，通常的取值为 MPI_
                   MIN、MPI_MAX 和 MPI_SUM
```

图 B-5 C 语言下 MPI 1.1 的主要集合操作函数通信例程（MPI_Barrier、MPI_Bcast 和 MPI_Reduce）

在图 B-6 和图 B-7 中，我们编写了一段用这三个函数的程序。在这个程序中，我们希望计算一个消息绕进程环传递的时间。我们不在这里讨论环形通信的函数，但是为了完整性，我们提供了图 B-8。对于任何并行程序来说，它的运行时间就是它最慢进程的运行时间。所以我们需要得到每个进程所花费的时间并且找到它们的最大值。我们使用 MPI 的标准计时函数：

```
double MPI_Wtime();
```

```
//
// Ring communication test.
// Command-line arguments define the size of the message
// and the number of times it is shifted around the ring:
//
// a.out msg_size num_shifts
//
#include "mpi.h"
#include <stdio.h>
#include <memory.h>

int main(int argc, char **argv) {
  int num_shifts = 0; // number of times to shift the message
  int buff_count = 0; // number of doubles in the message
  int num_procs = 0; // number of processes in the ring
  int ID; // process (or node) id
  int buff_size_bytes; // message size in bytes
  int i;

  double t0; // wall-clock time in seconds
  double ring_time; // ring comm time - this process
  double max_time; // ring comm time - max time for all processes
  double *x; // the outgoing message
  double *incoming; // the incoming message

  MPI_Status stat;

// Initialize the MPI environment
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &ID);
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

// Process, test, and broadcast input parameters
  if(ID == 0){
     if (argc != 3){
        printf("Usage: %s <size of message> <Num of shifts> \n",*argv);
        fflush(stdout);
        MPI_Abort(MPI_COMM_WORLD, 1);
     }
     buff_count = atoi(*++argv); num_shifts = atoi(*++argv);

     printf(": shift %d doubles %d times \n",buff_count, num_shifts);
     fflush(stdout);
  }

// Continued in the next figure
```

图 B-6 计算 ring 函数在环形进程中传递消息所用时间的程序（续见图 B-7）。这个程序返回使用通信时间最长的进程。ring 函数的编码与此例无关，但是它包含在图 B-8 中

```
// Continued from the previous figure

// Broadcast data from rank 0 process to all other processes
  MPI_Bcast (&buff_count, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast (&num_shifts, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Allocate space and fill the outgoing ("x") and "incoming" vectors.
  buff_size_bytes = buff_count * sizeof(double);

  x = (double*)malloc(buff_size_bytes);
  incoming = (double*)malloc(buff_size_bytes);
```

图 B-7 计算 ring 函数在环形进程中传递消息所用时间的程序（续图 B-6）

```
  for(i=0;i<buff_count;i++){
     x[i] = (double) i;
     incoming[i] = -1.0;
  }

// Do the ring communication tests.
  MPI_Barrier(MPI_COMM_WORLD);

  t0 = MPI_Wtime();
  /* code to pass messages around a ring */
  ring (x,incoming,buff_count,num_procs,num_shifts,ID);
  ring_time = MPI_Wtime() - t0;

// Analyze results
   MPI_Barrier(MPI_COMM_WORLD);

   MPI_Reduce(&ring_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0,
                                              MPI_COMM_WORLD);
   if(ID == 0){
     printf("\n Ring test took %f seconds", max_time);
   }
    MPI_Finalize();
}
```

图 B-7 （续）

MPI_Wtime() 返回自过去某一点到现在的时间（以秒为单位）。通常调用它两次来计算我们感兴趣的时间间隔。

这个程序一开始就像大多数 MPI 程序一样，声明，初始化 MPI，确定进程序号和进程数量。接着我们从命令行参数得到消息大小和消息绕环传递的次数。因为每个进程都会使用到这些值，所以调用 MPI_Bcast() 函数广播这些值。

然后给用于环形测试的输入 / 输出向量分配空间。为了得到一致的结果，所有进程必须在任意进程进入程序的计时部分前彻底初始化。这个工作可在进入计时部分前调用 MPI_Barrier() 函数得以保证。下一步，调用计时函数得到开始时间，环形测试开始，然后再次调用计时函数。两次计时函数的差就是这个进程将消息绕环传递所花的时间。

```
/*****************************************************************
NAME: ring
PURPOSE: This function does the ring communication, with the
         odd numbered processes sending then receiving while the even
         processes receive and then send.
         The sends are blocking sends, but this version of the ring
         test still is deadlock-free since each send always has a
         posted receive.
*****************************************************************/
#define IS_ODD(x) ((x)%2) /* test for an odd int */
#include "mpi.h"
 ring(
          double *x, /* message to shift around the ring */
          double *incoming, /* buffer to hold incoming message */
          int buff_count, /* size of message */
          int num_procs, /* total number of processes */a
          int num_shifts, /* numb of times to shift message */
          int my_ID) /* process id number */
{
```

图 B-8 用来绕进程环传递消息的函数。它避免了死锁，因为发送和接收是按照进程序号的奇偶性分开的

```
    int next; /* process id of the next process */
    int prev; /* process id of the prev process */
    int i;
    MPI_Status stat;
/*********************************************************************
** In this ring method, odd processes snd/rcv and even processes
   rcv/snd.
*********************************************************************/
    next = (my_ID +1 )%num_procs;
    prev = ((my_ID==0)?(num_procs-1):(my_ID-1));
    if( IS_ODD(my_ID) ){

        for(i=0;i<num_shifts; i++){
          MPI_Send (x, buff_count, MPI_DOUBLE, next, 3,
                    MPI_COMM_WORLD);
          MPI_Recv (incoming, buff_count, MPI_DOUBLE, prev, 3,
                    MPI_COMM_WORLD, &stat);
        }
    }
    else{
       for(i=0;i<num_shifts; i++){
          MPI_Recv (incoming, buff_count, MPI_DOUBLE, prev, 3,
                    MPI_COMM_WORLD, &stat);
          MPI_Send (x, buff_count, MPI_DOUBLE, next, 3,
                    MPI_COMM_WORLD);
       }
    }
}
```

图 B-8 （续）

对于一个 MPI 程序来说，运行的总时间是最慢处的理器所决定的。我们需要找到各个处理器所花费的最长时间。为此，只需要调用 MPI_Reduce() 函数，并使用 MPI_MAX 运算即可。

```
int MPI_Isend(outbuff, count, MPI_type, dest, tag, MPI_COMM, request);

int MPI_Irecv (inbuff, count, MPI_type, source, tag, MPI_COMM, request)

int MPI_Wait(request, status);

int MPI_Test(request, flag, status);
```

| | |
|---|---|
| inbuff、outbuff | 指向与 MPI_type 相兼容缓冲区的指针 |
| int count | buff 中包含指定类型的条目个数 |
| MPI_type | 定义在 mpi.h 中缓冲区里条目的类型 |
| int source | 发送消息进程的序号 |
| int tag | 消息的一个标签 |
| int dest | 接收最后归约输出结果的进程的序号 |
| MPI_COMM Comm | MPI 通信域，通常使用默认的 MPI_COMM_WORLD |
| MPI_Request request | 用于与通信交互的句柄 |
| int flag | 一个标志，如果消息完成它会变成非零的数字（逻辑值为真） |

图 B-9 非阻塞或异步通信函数

## B.5 高级的点对点消息传递

大多数 MPI 程序员仅仅使用标准的点对点传递函数。但是，为程序员能够控制更多通信

的细节，MPI 提供了额外的消息传递函数。可能这些高级消息传递函数中最重要的就是非阻塞（异步）通信函数。

非阻塞通信的优势是可以通过重叠计算和通信来用来降低并行开销。非阻塞通信函数可以直接返回，提供一个可以等待和查询的“请求句柄”。这些函数的语法可以在图 B-9 中看到。为了彻底了解这些函数是怎样使用的，我们提供了图 B-10 的程序。

在初始化 MPI 环境之后，为分配给域（U）以及用于和 U 边界通信的缓冲区（B 和 inB）分配内存。接着，对于每个进程通过从左向右计算 ID，从而建立起环形通信模式。每个迭代通过接收开始，进入主进程的循环：

280 ~ 284

```
MPI_Irecv(inB, N, MPI_DOUBLE, left, i, MPI_COMM_WORLD, &req_recv);
```

```
#include <stdio.h>
#include <mpi.h>
// Update a distributed field with a local N by N block on each process
// (held in the array U). The point of this example is to show
// communication overlapped with computation, so code for other
// functions is not included.

#define N 100 // size of an edge in the square field.
void extern initialize(int, double*);
void extern extract_boundary(int, double*, double*);
void extern update_interior(int, double*);
void extern update_edge(int,double*, double*);

int main(int argc, char **argv) {
  double *U, *B, *inB;
  int i, num_procs, ID, left, right, Nsteps = 100;
  MPI_Status status;
  MPI_Request req_recv, req_send;

// Initialize the MPI environment
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &ID);
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

// allocate space for the field (U), and the buffers
// to send and receive the edges (B, inB)
  U = (double*)malloc(N*N * sizeof(double));
  B = (double*)malloc(N * sizeof(double));
  inB = (double*)malloc(N * sizeof(double));

// Initialize the field and set up a ring communication pattern
  initialize (N, U);
  right = ID + 1; if (right > (num_procs-1)) right = 0;
  left = ID - 1; if (left < 0 ) left = num_procs-1;

// Iteratively update the field U
  for(i=0; i<Nsteps; i++){
     MPI_Irecv(inB, N, MPI_DOUBLE, left, i, MPI_COMM_WORLD, &req_recv);
     extract_boundary(N, U, B); //Copy the edge of U into B
     MPI_Isend(B, N, MPI_DOUBLE, right, i, MPI_COMM_WORLD, &req_send);
     update_interior(N, U);
     MPI_Wait(&req_recv, &status); MPI_Wait(&req_send, &status);
     update_edge(ID, inB, U);
  }
  MPI_Finalize();
}
```

图 B-10　用非堵塞通信迭代更新域的程序，此算法只需要从左到右移动消息

一旦系统建立起用于保存从左边来的消息资源，此函数变马上返回。句柄 req_recv 提供了一种关于通信状态的询问机制。接着域的边界被抽取出来然后发送到右邻居去。

```
MPI_Isend(B, N, MPI_DOUBLE, right, i, MPI_COMM_WORLD, &req_send);
```

当通信发生的时候，程序更新了域的内层（内的意思是这部分的更新不需要邻居进程的信息）。当工作完成后，每个进程必须等待，直到所有通信完成。

```
MPI_Wait(&req_send, &status);
MPI_Wait(&req_recv, &status);
```

通信结束之后，该领域的边缘被更新，程序继续进行下一次迭代。

MPI 中另一项减少并行花销的技术是持续通信。当一个问题重复使用某种通信模式的时候，就会考虑这种技术。这个想法是建立一次通信接着让他传递多次实际消息。持续通信中使用的函数有：

```
MPI_Send_init(outbuff, count, MPI_type, dest, tag, MPI_COMM, &request);
MPI_Recv_init(inbuff, count, MPI_type, src, tag, MPI_COMM, &request);
MPI_Start(&request);
MPI_Wait(&request, &status);
```

MPI_Send_init 和 MPI_Recv_init 用来建立通信。这两个函数将会返回一个请求句柄用来操作实际通信的事件。这个通信将会因为 MPI_Start 的调用而初始化，在这一点上，这个进程都是准备好继续进行任何计算的。通信完成之前，当没有后续的工作可以做的时候，进程可以调用 MPI_Wait 函数进行等待。对于在图 B-8 中表达的环形通信模式用持续通信的函数见图 B-11。

除了非堵塞通信和持续通信之外，MPI 定义了几个通信模式对应不同的通信缓冲区发送方式。

- **标准通信模式**（MPI_Send）。标准的 MPI 发送；直到发送缓冲区被清空并可以重用的时候，这个发送过程才算完成。
- **阻塞通信模式**（MPI_Ssend）。发送直到一个匹配的接收已经发送后才算完成。这使得可以将通信作为一个对同步事件。
- **缓冲通信模式**（MPI_Bsend）。用户提供缓冲空间用于缓冲消息。这种发送会随着发送到缓冲区的消息复制到系统缓冲区而结束。
- **就绪通信模式**（MPI_Rsend）。发送将假设一个匹配的接收已经发送（否则程序出错），因此立即传送消息。在一些系统中，就绪通信模式会更加有效率。

本书中大多数 MPI 使用的是标准模式。第 6 章使用同步模式实现互斥。其他模式的更多细节可以在 MPI 规范 [Mesb] 中查到。

```
/*****************************************************************
NAME: ring_persistent
PURPOSE: This function uses the persistent communication request
        mechanism to implement the ring communication in MPI.
*****************************************************************/

#include "mpi.h"
#include <stdio.h>
```

图 B-11　使用持续通信使得消息绕环传递的函数

```
ring_persistent(
          double *x, /* message to shift around the ring */
          double *incoming, /* buffer to hold incoming message */
          int buff_count, /* size of message */
          int num_procs, /* total number of processes */
          int num_shifts, /* numb of times to shift message */
          int my_ID) /* process id number */
{
   int next; /* process id of the next process */
   int prev; /* process id of the prev process */
   int i;
   MPI_Request snd_req; /* handle to the persistent send */
   MPI_Request rcv_req; /* handle to the persistent receive */
   MPI_Status stat;

/*****************************************************************
** In this ring method, first post all the sends and then pick up
** the messages with the receives.
*****************************************************************/
   next = (my_ID +1 )%num_procs;
   prev = ((my_ID==0)?(num_procs-1):(my_ID-1));

   MPI_Send_init(x, buff_count, MPI_DOUBLE, next, 3,
                              MPI_COMM_WORLD, &snd_req);
   MPI_Recv_init(incoming, buff_count, MPI_DOUBLE, prev, 3,
                              MPI_COMM_WORLD, &rcv_req);
   for(i=0;i<num_shifts; i++){

     MPI_Start(&snd_req);
     MPI_Start(&rcv_req);

     MPI_Wait(&snd_req, &stat);
     MPI_Wait(&rcv_req, &stat);

   }
}
```

图 B-11 （续）

## B.6 MPI 和 FORTRAN

正式的 MPI 定义有 C 和 FORTRAN 两种语言绑定。对 MPI 和 Java 的绑定也有定义[BC00、Min97、BCKL98、AJMJS02]。到此为止，我们只关注了 C 语言。但是在一些简单规则的基础上，把 C 语言翻译到 Fortran 语言是非常容易的。

- Fortran 中，头文件 `mpif.h` 包括常量、错误代码等。
- MPI 例程在两种语言中基本上都是一个名字。但是在 C 语言中 MPI 函数是区分大小写的，Fortran 则不区分大小写。
- 除了计时例程的绝大部分情况下，Fortran 使用子例程，但是 C 语言使用函数。
- C 函数的参数的数据类型和 Fortran 子例程的参数的数据类型能够明显对应。有一种额外的参数在 Fortran 子例程中会经常见到，这是一个整数参数：`ierr`，它保存 MPI 错误返回码。

比如，我们将 MPI 的归约函数的 C 和 Fortran 版本在图 B-12 中对比。

在C中，一个归约操作由下列函数完成

```
int MPI_Reduce(inbuff,outbuff,count,MPI_type,OP,dest,COMM);

inbuff、outbuff        指向一个兼容MPI_type的缓冲区的指针
int count              缓冲区中包含的指定数据类型的条目数量
MPI_type               缓冲区中条目的类型，其在mpi.h中定义
OP                     用于归约操作的运算，其在mpi.h中定义。通常的
                       值是MPI_MIN、MPI_MAX和MPI_SUM
int dest               用于保存最终输出结果的进程号
MPI_COMM Comm          MPI通信域，通常的选择是默认值MPI_COMM_
                       WORLD
```

类比于Fortran中的以下子例程：

```
    subroutine MPI_REDUCE(inbuff, outbuff, count, MPI_
type, OP, dest, COMM, ierr)

inbuff、outbuff        恰当尺寸和类型的数组
integer count          缓冲区中包含的指定数据类型的条目数量
MPI_type               缓冲区中条目的类型，其在mpi.h中定义
OP                     用于归约操作的运算，其在mpi.h中定义。通常的
                       值是MPI_MIN、MPI_MAX和MPI_SUM
integer dest           用于保存最终输出结果的进程号
MPI_COMM Comm          MPI通信域，通常的选择是默认值MPI_COMM_
                       WORLD
interger ierr          一个用于保存错误码的整数，其在mpi.h中定义
```

图B-12 C版本和Fortran版本MPI 1.1归约例程的比较

不透明对象（如MPI_COMM或者MPI_Request）在Fortran中是一个INTEGER类型（例外的是布尔变量，其在Fortran中是LOGICAL）。

一个简单的MPI Fortran程序如图B-13所示。这个程序展示了Fortran中MPI基本的启动例程和结束子例程的工作原理。直接类比C语言的MPI会在图B-13中表达得非常清楚。基本上，如果一个程序员能够在一种语言中理解MPI，那么他能够理解另外一种语言的MPI。基本结构在两种语言中是相同的。但是，如果程序员在混合Fortran-MPI和C-MPI编程的时候一定要小心，因为MPI规范没有保证两种语言间的互操作性。

```
program firstprog
include "mpif.h"
integer ID, Nprocs, ierr

call MPI_INIT( ierr )
call MPI_COMM_RANK ( MPI_COMM_WORLD, ID, ierr)
call MPI_COMM_SIZE ( MPI_COMM_WORLD, Nprocs, ierr )

print *, "Process ", ID, " out of ", Nprocs

call MPI_FINALIZE( ierr )

end
```

图 B-13 简单的 MPI Fortran 程序。此程序用于输出各个进程的 ID 和计算中进程的数量

285 ~ 289

## B.7 小结

到目前为止，MPI 是最流行的并行编程 API。它经常叫做并行编程的“汇编代码”。MPI 的底层结构非常匹配 MIMD 模型的并行计算机。这就让 MPI 程序员能够精准控制并行计算如何展开，如何写出非常高效的程序。可能更为重要的是，这样程序员可以写出可移植的并行程序，它们运行在共享内存计算机、大规模并行超级计算机、集群甚至网格上。

学习 MPI 是非常麻烦的事情。它非常庞大，MPI 1.1 中有 125 个不同的函数。MPI 这样大的规模确实使它变得很复杂，但是多数程序员避免了这种复杂性，仅仅使用 MPI 的一个小子集。很多并行程序只需要简单的 6 个函数就能写出来：`MPI_Init`、`MPI_Comm_Size`、`MPI_Comm_Rank`、`MPI_Send`、`MPI_Recv` 和 `MPI_Finalize`。我们可以从 [Pac96]、[GLS99] 和 [GS98] 中得到更多关于 MPI 的优质信息。不同版本的 MPI 对大多数计算机系统是可用的，通常他们都以开源软件的形式在网络上发布。最常用的 MPI 版本是 LAM/MPI [LAM] 和 MPICH [MPI]。

290

附录 C

Patterns for Parallel Programming

# Java 并发编程简介

Java 是一种面向对象的编程语言，它对共享内存程序的并发表达提供了语言支持，Java 对于多态的支持可用来编写编程框架以直接支持本书描述的一些模式。该框架为这些模式提供基础架构，应用程序程序员可添加包含特定应用程序代码的子类。在 4.8 节的示例部分中可找到这样的一个例子。Java 程序通常被编译成称为 Java 字节码的中间语言，然后在目标机器上编译和 / 或解释 Java 字节码，因此 Java 程序享有高度可移植性。

Java 也可用于分布式计算，它的标准库为分布式系统中进程间通信提供了几种机制支持。第 6 章给出了一个简单的阐述。另外，存在一个不断增长的包集合，这些包使得并发性和进程间通信能够以一种比利用语言和核心包所提供的工具更高的水平来表达。当前，利用 Java 工具开发的几类最常用的并发性应用程序包括多线程服务器端应用程序、图形用户界面和一些因为使用了不同位置的数据和（或）资源而自然分布的程序。

利用 Java 的当前实现所获得性能并不比利用高性能科学计算中通常所使用的一些编程语言更好。然而，Java 无处不在的特性和可移植性，以及对并发性的语言支持，使得它成为一种重要的平台。对于许多人来说，Java 程序很可能是他们的第一个并发程序。另外，不断提高的编译器技术和库函数应当能够在将来降低性能差别。

本附录描述 Java 2 1.5 ⊖ [Java] ，与早期版本相比，它在编程语言和标准库中添加了一些新的功能。在编程语言中所添加的内容包括对泛型类型和自动装箱操作（自动装箱提供了基本数据类型和对象类型间的自动转换，如 `int` 和 `Integer`）的支持，并增强了 `for` 循环。图 C-1 展示了使用泛型类型的程序，这是本书唯一一次描述具体语言。

```
/*class holding two objects of arbitrary type providing swap and
accessor methods. The generic type variable name is enclosed in
angle brackets*/

class Pair<Gentype>
//Gentype is a type variable
{ private Gentype x,y;
  void swap(){Gentype temp = x; x=y; y=temp;}
  public Gentype getX(){return x;}
  public Gentype getY(){return y;}
  Pair(Gentype x, Gentype y){this.x = x; this.y = y;}
  public String toString(){return "(" + x + "," + y + ")";}

}

/*The following method, defined in some class, instantiates two
Pair objects, replacing the type variable in angle brackets with
```

图 C-1　这个类中包含了一对任意类型的对象。如果没有泛型类型，这将通过声明 `x` 和 `y` 为 `Object` 类型来完成，并需要强制转换 `getX` 和 `getY` 的返回值类型。通过使用泛型，不仅使代码更简洁，而且可使一些类型错误能够被编译器发现，而不是在运行时抛出一个 `ClassCastException` 异常

⊖ 在本书截稿时，可以获得的 Java 版本是 J2SE 1.5.0 Beta 1。

```
the actual type the object should contain. */

   public static void test()
   { /*create a Pair holding Strings.*/
      Pair<String> p1 = new Pair<String>("hello","goodbye");

      /*create a Pair holding Integers. Autoboxing automatically
      creates Integer objects from the given int parameters,
      saving the programmer from writing
      new Pair<Integer>(new Integer(1), new Integer(2));*/
      Pair<Integer> p2 = new Pair<Integer>(1,2);

      /*do something with the Pairs.*/
      System.out.println(p1); p1.swap(); System.out.println(p1);
      System.out.println(p2); p2.swap(); System.out.println(p2);
   }
```

图 C-1 （续）

Java 2 1.5 通过在 java.lang 和 java.util 包中做了几处改动，并引入了一些新包，如 java.util.concurrent、java.util.concurrent.atomic 和 java.util.concurrent.locks，为并发编程提供了重要支持。Java 2 1.5 还注册了一个新的、更精确的内存模型规范。[JSRb、JSRc] 中描述了用于并发编程的工具，[JSRa] 中描述了新的内存模型。

本附录无法涵盖所有的 Java 知识。因此，我们假设你已经对这门语言有一定的了解，并着重介绍与共享内存并行编程最相关的一些内容，包括 Java 2 1.5 中引入的一些新的特征。在 [AGH00、HC02、HC01] 中可以找到关于 Java 和它的库（Java 2 1.5 之前的版本）的介绍。在 [Lea00a] 中很好地讨论了 Java 中的并发编程。

## C.1 线程的创建

在 Java 中，一个应用程序至少具有一个执行 main 方法的线程。可以通过实例化并启动 thread 对象创建一个新线程；该线程要执行的代码由 run 方法提供，后面将详细介绍这一点。当一个线程的 run 方法返回时，该线程将终止。线程可以是普通类型，也可以是一个守护线程。在一个应用程序中，每一个线程与其他线程之间都是独立运行的，当所有非守护线程终止时，应用程序也将终止。

根据 Java 通常的作用域规则，一个线程可以访问其 run 方法可见的所有变量。这样，通过使一些变量对多个线程可见，能够使存储器在这些线程间共享。通过封装变量，可使内存不被其他线程访问。另外，一个变量可以声明为 final 类型，一旦初始化，它的值将不再改变。如果已经正确初始化，final 类型的变量可以被多个线程访问，而不需要同步（但将一个引用声明为 final 类型，并不能保证所引用对象的不变性；使用时必须小心确保它是线程级安全的）。

有两种不同方法来指定线程将要执行的 run 方法。一种是扩展 Thread 类，并重写 run 方法。然后，实例化子类的一个对象，并调用子类所继承的 start 方法。更通用的方法是使用以一个 Runnable 对象作为参数的构造函数来实例化 Thread 对象。像前面的方法一样，调用 Thread 对象的 start 方法来启动线程的执行。Runnable 接口包含一个 public void run 方法；新建线程将执行 Runnable 对象的 run 方法。因为 run 方法没

有参数，所以信息通常通过 Thread 子类或 Runnable 的数据成员传递给线程。这些通常在构造函数中设置。

在下面的示例中，ThinkParallel 类实现了 Runnable 接口（即它声明它实现这个接口，并提供一个 public void run() 方法）。main 方法创建并启动了 4 个 Thread 对象，每个 Thread 对象传递一个 ThinkParallel 对象，该对象的 id 字段已经设置为循环计数器 i 的值。这将导致 4 个线程被创建，每个线程执行 ThinkParallel 类的 run 方法。

```
class ThinkParallel implements Runnable
{
    int id; //thread-specific variable containing thread ID

    /*The run method defines the thread's behavior*/
    public void run()
    { System.out.println(id + ": Are we there yet?");
    }

    /*Constructor sets id*/
    ThinkParallel(int id){this.id = id;}

    /*main method instantiates and starts the threads*/
    public static void main(String[] args)

    { /*create and start 4 Thread objects,
        passing each a ThinkParallel object
      */

    for(int i = 0; i != 4; i++)
        { new Thread(new ThinkParallel(i)).start(); }
    }
}
```

图 C-2 该程序创建了 4 个线程，在线程的构造函数中传递一个 Runnable 对象。线程专用的数据存储在 Runnable 对象的一个字段中

### C.1.1 匿名内部类

几个示例展示了一种常用编程习惯：使用一个匿名类在创建它们的地方定义 Runnables 或者 Threads。这通常使得阅读源代码更加方便，并且避免了文件和类的混乱。我们在图 C-3 中演示了如何使用这种编程习惯重新编写图 C-2 中的程序。不能在匿名类中提及一个方法的局部变量，除非该变量被声明为 final 类型（该变量被赋值后不可改变）。这是引入看似冗余的变量 j 的原因。

```
class ThinkParallelAnon {

  /*main method instantiates and starts the threads*/

  public static void main(String[] args)

  { /*create and start 4 Thread objects,
      passing each a Runnable object defined by an anonymous class
    */
```

图 C-3 与图 C-2 中的程序相似的程序，但是这个程序使用了匿名类来定义 Runnable 对象

```
      for(int i = 0; i != 4; i++)
      { final int j = i;
        new Thread( new Runnable() //define Runnable objects
                                   // anonymously
                    { int id = j; //references
                      public void run()
                      { System.out.println(id + ":
                                          Are we there yet?");}
                    }

                ).start();

      }
    }
}
```

图 C-3 （续）

### C.1.2 Executor 和工厂方法

java.util.concurrent 包提供了 Executor 接口和它的子接口 ExecutorService，以及 ExecutorServices 的几个实现。Executor 执行 Runnable 对象，同时隐藏了线程创建和调度的细节[⊖]。这样，可以实例化一个 Executor 对象，并调用 Executor 的 execute 方法来执行 Runnable，而不是实例化一个 Thread 对象并给它传递一个 Runnable 对象以执行一个任务。例如，ThreadPoolExecutor 管理一个线程池，并使用其中的一个线程执行提交的 Runnable。虽然实现了这些接口的类提供了一些可调整参数，但是 Executor 类提供了通过创建了各种 ExecutorService 来预配置一些最常用情形的工厂方法（factory method）。例如，工厂方法 Executors.newCachedThreadPool() 返回一个具有一个未绑定线程池并且能够自动回收线程的 ExecutorService。如果可以获得线程池中的一个线程，将尝试使用该线程，如果不能获得，则创建一个新线程。在一定时间后，空闲线程将被清除出线程池。另外一个示例是 Executors.newFixedThreadPool(int nThread)，它创建一个包含 nThreads 个线程的固定大小线程池。该线程池包含一个未绑定的用于存放等待任务的队列。其他的配置也是可以的，包括使用这里没有描述的其他工厂方法，或者直接实例化一个实现 ExecutorService 的类并手动调整参数。

291 ~ 294

通过重写前面的程序，图 C-4 展示了这样一个示例。需要注意的是，为了采用一种不同的线程管理策略，我们将仅需要调用一个不同的工厂方法来实例化 Executor。代码的其他部分保持不变。

Runnable 接口中的 run 方法不返回值，并且不能抛出异常。为了纠正这个缺陷，引入了 Callable 接口，该接口包含一个名为 call 的方法，该方法能够抛出异常并返回一个结果。该接口的定义利用了对泛型类型的新支持。通过使用实现了 ExecutorService 接口的对象的 submit 方法，Callable 能够安排执行。submit 方法返回一个 Future 对象，表示异步计算的结果。为了等待计算的完成，并且获得结果，Future 对象提供了一些用于

⊖ ExecutorServices 接口提供了一些用于管理线程终止和支持 Future 的额外方法。在 java.util.concurrent 包中 Executor 的实现也实现了 ExecutorService 接口。

检测计算是否完成的方法。包含在尖括号中的类型指定该类将被声明为这种类型。

```
import java.util.concurrent.*;

class ThinkParalleln implements Runnable {

    int id; //thread-specific variable containing thread ID

    /*The run method defines the tasks behavior*/
    public void run()
    { System.out.println(id + ": Are we there yet?");
    }

    /*Constructor sets id*/
    ThinkParalleln(int id){this.id = id;}

    /*main method creates an Executor to manage the tasks*/
    public static void main(String[] args)

    { /*create an Executor using a factory method in Executors*/
      ExecutorService executor = Executors.newCachedThreadPool();

      // send each task to the executor
      for(int i = 0; i != 4; i++)
        { executor.execute(new ThinkParalleln(i)); }


      /*shuts down after all queued tasks handled*/
      executor.shutdown();

    }
}
```

图 C-4 本程序使用了一个 ThreadPoolExecutor 而不是直接创建线程

图C-5中给出了一个主线程将一个匿名的Callable传递给一个Executor的代码段。Executor在另一个线程中安排call方法的执行。同时，初始线程完成某些其他工作，然后使用get方法获得Callable执行的结果。如果需要，主线程将在get方法处阻塞，直到获得可用的结果。

```
/*execute a Callable that returns a Double.
The notation Future<Double> indicates a (generic) Future that
has been specialized to be a <Double>.
*/
Future<Double> future = executor.submit(
     new Callable<Double>() {
         public Double call() { return result_of_long_computation; }
 });
 do_something_else(); /* do other things while long
   computation proceeds in another thread */
 try {
   Double d = (future.get()); /* get results of long
     computation, waiting if necessary*/
 } catch (ExecutionException ex) { cleanup(); return; }
```

图 C-5 代码段说明 Callable 和 Future 的使用

## C.2 原子性、内存同步和 volatile 关键字

Java 虚拟机（JVM）规范要求除 long 和 double 之外的所有基本类型的读和写操作必

须是原子性的。这样，一条类似于 done=true 这样的语句将不与其他线程相互干扰。遗憾的是，像第 6 章所解释的一样，我们不仅需要原子性，而且需要确保写操作对其他线程是可见的，其他线程将访问该变量的最新值。在 Java 中，一个变量被标识为 volatile 类型将保证所有访问都会被内存同步。另外，应用于 long 和 double 类型的 volatile 关键字也保证了原子性。

java.util.concurrent.atomic 包提供了对原子性和内存同步的扩展支持。例如，该包包含大量在其类型中提供 compareAndSet 原子操作的类。在许多系统中，这样的操作可以实现为单条机器指令。这样该包可以提供其他操作，例如，原子递增。该包也提供每个数组元素都具有 volatile 语义的一些数组类[⊖]。仅当对一个对象的关键更新被限制为对单个变量的更新时，java.util.concurrent.atomic 包中的类才有意义。因此，它们通常被用作为较高级别构造的构造块。

下面的代码使用了 AtomicLong 类的 getAndIncrement 方法实现了一个线程安全的序列发生器（sequencer）。这个 getAndIncrement 方法自动获得变量的值，递增变量，并返回初始值。这样的序列发生器可在某些主 / 从进程设计中实现一个任务队列。

```
class Sequencer
{
  private AtomicLong sequenceNumber = new AtomicLong(0);
  public long next() { return sequenceNumber.getAndIncrement(); }
}
```

## C.3 同步块

当变量可以被多个线程访问时，必须十分谨慎以确保不出现损坏数据的争用条件。Java 提供一种称为同步块的构造，允许程序员确保对共享变量的互斥访问。同步块也可以用作为隐式的内存栅栏，就像 6.3 节所描述的一样。

Java 程序中的每一个类都是 Object 类的一个直接或间接子类。每一个 Object 实例都隐式地包含一个锁。一个同步块始终与一个对象相关联：在一个线程能够进入一个同步块之前，它必须获得与该对象相关联的锁。当线程离开同步块时，无论是正常还是因为抛出异常而离开，都释放锁。在同一时间内，至多只有一个线程能够拥有锁，因此同步块能够确保数据的互斥访问。同时，它们也可用于同步内存。

295 ~ 297

声明一个同步块的代码如下所示：

```
synchronized(object_ref){...body of block....}
```

大括号限定了同步块。用于获取和释放锁的代码由编译器产生。

假设我们在 ThinkParallel 类中添加了一个 static int count 变量，在每个线程输出它的消息之后递增该变量。该变量为静态变量，因此每个类（而不是每个对象）中都存在一个这样的变量，它对于所有线程都可见并进而被共享。为了避免争用条件，可以将 count 变量放置到一个同步块中[⊜]。为了提供保护，所有线程必须使用相同的锁，因此我们

⊖ 在 Java 语言中，将一个数组声明为 volatile 类型，仅使得数组的引用而不是每个元素为 volatile 类型。

⊜ 当然，对于这种特殊情形，可以改用 java.util.concurrent.atomic 包中定义的原子变量。

使用与该类本身相关联的对象。对于任意类 X，X.class 是对代表类 X 的唯一对象的一个引用，因此可以按如下方式编写代码：

```
public void run()
{ System.out.println(id + ": Are we there yet?");
  synchronized(ThinkParallel.class){count++;}
}
```

需要强调的是，仅仅那些与相同对象相关联的同步块才相互排斥。与不同对象相关联的两个同步块可以并发地执行。例如，在编写前面的代码时，一种常见的编程错误为：

```
public void run()
{ System.out.println(id + ": Are we there yet?");
  synchronized(this){count++;} //WRONG!
}
```

298

在这个错误版本中，每个线程将同步于与“自己”或者 this 对象相关联的锁。这将意味着每个线程在进入同步块时将获得一个不同的锁（与该线程对象本身相关联的锁），因此它们不会相互排斥。另外，同步块不约束线程访问非同步块中的共享变量的行为。程序员必须仔细确保所有共享变量受到合理的保护了。

Java 为通用情形提供了一种特别的语法，其中整个方法体位于与 this 对象相关联的一个同步块中。在这种情形中，synchronized 关键字用于修改方法的声明。即：

```
public synchronized void updateSharedVariables(...)
{....body.... }
```

可简写为：

```
public void updateSharedVariables(...)
{ synchronized(this){....body....} }
```

## C.4 等待和通知

一个线程有时需要检测是否满足某些条件。如果满足条件，则线程应当执行某种操作；如果条件不满足，则它应当继续等待，直到某些其他线程建立了这种条件。例如，一个线程可能要查看一个缓冲区是否包含一项记录。如果包含，则移除该项记录；如果它不包含，则线程继续等待，直到其他线程插入这项记录。检测条件和执行动作都必须以原子方式进行。否则，一个线程可以检测条件（即，检测缓冲区非空），另一个线程可以伪造这个条件（通过移除仅有的记录），于是第一个线程将执行一个依赖于这个（不再满足）条件的动作，并且错置程序状态。因而，检测条件和执行动作需要放置在同一个同步块中。另一方面，线程在等待时，不能拥有与同步块相关联的锁。因为这样将阻塞其他线程的访问，从而阻止条件的建立。当线程正在等待一个条件时，需要释放锁；在条件满足后，在重新检测条件并在执行它的动作之前，线程将重新获取锁。传统的监控器 [Hoa74] 都主张处理这种情形。Java 提供了一些相似的工具。

299

Object 类以及前面提到的锁，也包含一个用作条件变量的隐式等待集合。Object 类提供了多个 wait 方法版本，这些 wait 方法使得主调线程隐式地释放锁，并将线程本身添加到等待集合中。等待集合中的线程不符合被调度运行的条件，将被挂起。

wait 方法的基本使用方式如图 C-6 所示：线程获取与 lockObject 相关联的锁，并检测 condition 是否满足。如果不满足条件，则执行 while 循环体（即 wait 方法）以及 wait 方法。这使得锁被释放，挂起线程并将其放置到属于 lockObject 的等待集合中。如果满足条件，线程会执行 action 并离开同步块。在离开同步块时，锁会被释放。

```
synchronized(lockObject)
{ while( ! condition ){ lockObject.wait();}
   action;
}
```

图 C-6 wait 方法的基本使用方式。因为 wait 会抛出异常 InterruptedException，所以需要将它封装在一个 try-catch 块中，但这里省略了

线程离开等待集合会用下面三种方式中的一种。第一，Object 类的 notify 方法和 notifyAll 方法将唤醒该对象等待集合中相应的一个或所有线程。这两个方法倾向于被建立等待条件的线程所调用。被唤醒线程离开等待集合并在继续执行之前，重新获取锁。可以不使用任何参数或超时的值来调用 wait 方法。使用计时 wait 方法的线程（即，给定线程一个超时的值）可以像前面描述的一样通过通知唤醒，或者在超时的值到期后被系统在某一点处所唤醒。一旦被唤醒，线程将重新获取锁，并继续正常地执行。遗憾的是，无法确定一个线程会被通知还是超时唤醒。线程离开等待集合的第三种方式是中断。这样会导致抛出一个 InterruptedException 异常，于是线程中的控制流将根据处理异常的正常规则进行异常处理。

现在，我们继续对图 C-6 中的情况进行描述。线程在某一点处等待并被唤醒。当线程被某个在 lockObject 上执行 notify 或者 notifyAll 方法的其他线程所唤醒时（或者被一个超时），将从等待集合中移除线程。在某个点上，将调度执行它，并且将试图重新获取与 lockObject 相关联的锁。在重新获取锁后，线程将重新检测条件，如果不满足条件将释放锁并再次等待，如果条件满足，则执行 action 而不释放锁。 300

程序员负责确保在条件已经建立之后适当通知等待线程，不正确的处理会造成程序停止运行。下面的代码展示了在程序建立条件后，使用 notifyAll 通知所有线程的方法：

```
synchronized(lockObject) {
  establish_the_condition;
  lockObject.notifyAll()
}
```

在标准用法中，wait 方法在一个 while 循环体中调用。这确保在执行动作之前将重新检查条件，并且大大增强了程序的健壮性。我们不应当通过将 while 循环更改为一条 if 语句来节省少量的 CPU 周期，另外，while 循环能够确保额外的 notify 方法不会引起错误。因此，我们可以在任意可能建立条件的地方使用 notifyAll 方法，然后通过仔细分析来消除一些不必要的 notifyAll 方法。在某些程序中，可以使用 notify 替换 notifyAll，可能会使程序的性能得以提高。但是，这些优化应当谨慎地进行。在 5.9 节中有一个这样的示例。

## C.5 锁

当按前面章节中所描述的方式直接使用时，同步块以及 wait 方法和 notify 方法在语

义方面具有某些缺陷。也许最严重的问题是，不存在对于隐式锁相关联的状态信息的访问。这意味着线程在尝试获取锁之前无法确定该锁是否可用。另外，等待与一个同步块相关联的锁的线程阻塞不能够中断[⊖]。另一个问题是，与每一个锁相关的（隐式）条件变量只有一个。因此，等待不同待建立条件的线程共享同一个等待集合，而且利用 notify 方法可能会唤醒错误的线程（于是强制使用 notifyAll 方法）。

因为这个原因，在以前很多 Java 程序员实现它们自己的锁原语，或者使用第三方包[⊜]，例如 util.concurrent [Leal]。现在，java.util.concurrent.locks 包提供 ReentrantLock 锁，它们与同步块相似，但是具有一些额外的功能。该锁必须显式地初始化，如下：

301
```
//instantiate lock
private final ReentrantLock lock = new ReentrantLock();
```

应当用在一个 try-catch 模块中，例如：

```
//critical section
lock.lock(); // block until lock acquired
try { critical_section }
finally { lock.unlock(); }
```

其他的方法允许获取关于锁的状态信息。这些锁在便利的语法和编译器的某些支持方面进行权衡（程序员不可能忘记释放与一个同步块相关联的锁），以获得更好的灵活性。

另外，该包提供了新的 Condition 接口的实现，该接口实现一个条件变量。这使得多个条件变量关联于单个锁。通过调用锁的 newCondition 方法可以获得与一个锁相关联的一个 Condition。与 wait、notify 和 notifyAll 类似的是 await、signal 和 signalAll。在图 C-7 中展示了一个使用这些新类来实现一个共享队列（就像 5.9 节中所描述的一样）的示例。

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

class SharedQueue2 {
   class Node
   { Object task;
      Node next;

      Node(Object task)
      {this.task = task; next = null;}
   }

   private Node head = new Node(null);
   private Node last = head;

   Lock lock = new ReentrantLock();
   final Condition notEmpty = lock.newCondition();
```

图 C-7 使用 Lock 和 Condition 的 SharedQueue2（参考 5.9 节）的一个版本，该版本并没有使用 wait 和 notify 同步块

⊖ 这将在下一节深入讨论。

⊜ 如果一个锁可以被同一个线程多次获取而不产生死锁，则该锁是可重复进入的。

```
    public void put(Object task)
    { //cannot insert null
       assert task != null: "Cannot insert null task";
       lock.lock();
       try{ Node p = new Node(task); last.next = p; last = p;
            notEmpty.signalAll();
       } finally{lock.unlock();}
    }
    public Object take()
    { //returns first task in queue, waits if queue is empty
       Object task = null;
       lock.lock();
       try {
       while (isEmpty())
          { try{ notEmpty.await(); }
            catch(InterruptedException error)
            { assert false:"sq2: no interrupts here";}
          }
       Node first = head.next; task = first.task; first.task = null;
       head = first;
       } finally{lock.unlock();}
       return task;
    }
    private boolean isEmpty(){return head.next == null;}
}
```

图 C-7 （续）

## C.6 其他同步机制和共享数据结构

与 Java 相比，OpenMP 具有与同步块相似的结构（锁和临界区），但缺少与 wait 和 notify 相似的功能。这是因为 OpenMP 提供的是在并行编程中常用的更高级别的结构，而不是像在 Java 中使用的更加通用的方法。我们可以非常容易地使用一些可用的功能来实现大量高级结构。另外，java.util.concurrent 包提供几种较高级的同步原语和共享数据结构。这些同步原语包括 CountDownLatch（一个简单的单用途栅栏，它将阻塞线程直到给定数目的线程到达该栅栏）、CyclicBarrier（一个周期栅栏，当线程通过之后将自动重置，这样方便多次使用，例如，在循环中）和 Exchanger（它使两个线程在一个同步点处交换对象）。6.3.2 节对 CountDownLatch 和 CyclicBarrier 做了详细的讨论。

在图 C-8 中展示了一个基于循环的非常简单的程序，同时在图 C-9 中展示了一个并行版本。这与附录 A 中的示例是相似的。

```
class SequentialLoop {
   static int num_iters = 1000;
   static double[] res = new double[num_iters];
   static double answer = 0.0;
   static void combine(int i){.....}
   static double big_comp(int i){.....}
   public static void main(String[] args)
   { for (int i = 0; i < num_iters; i++){ res[i] = big_comp(i); }
     for (int i = 0; i < num_iters; i++){ combine(i);}
     System.out.println(answer));
   }
}
```

图 C-8 一个类似于图 A-5 中基于循环的简单串行程序

302 ~ 303

```
import java.util.concurrent.*;

class ParallelLoop {

   static ExecutorService exec;
   static CountDownLatch done;
   static int num_iters = 1000;
   static double[] res = new double[num_iters];
   static double answer = 0.0;

   static void combine(int i){.....}
   static double big_comp(int i){.....}

   public static void main(String[] args) throws InterruptedException
   { /*create executor with pool of 10 threads */
       exec = Executors.newFixedThreadPool(10);

       /*create and initialize the countdown latch*/
       done = new CountDownLatch(num_iters);

       long startTime = System.nanoTime();
       for (int i = 0; i <.num_iters; i++)
       { //only final local vars can be referenced in an anonymous class
         final int j = i;
         /*pass the executor a Runnable object to execute the loop
         body and decrement the CountDownLatch */
         exec.execute(new Runnable(){
           public void run()
           { res[j] = big_comp(j);
             done.countDown(); /*decrement the CountDownLatch*/
           }
         });
       }
       done.await(); //wait until all tasks have completed

       /*combine results using sequential loop*/
       for (int i = 0; i < num_iters; i++){ combine(i); }
       System.out.println(answer);

       /*cleanly shut down thread pool*/
       exec.shutdown();
   }
}
```

图 C-9 这个程序展示了图 C-8 中串行程序的一个并行版本，其中 big-comp 循环的每次迭代都是一个独立的任务。程序使用了一个包含 10 个线程的线程池来执行这些任务，使用了 CountDownLatch 以确保所有任务在执行（仍然是串行的）组合结果的循环之前已经完成

java.util.concurrent 包也提供了共享队列模式的几种实现和一些线程安全的 Collection 类，包括 ConcurrentHashMap、CopyOnWriteArrayList 和 CopyOnWriteArraySet。

## C.7 中断

中断是线程的一个状态。可以使用 interrupt 方法来中断线程。这个方法将线程的中
304 断状态设置为“被中断”。

如果线程被挂起（也就是说，线程执行了 wait、sleep、join 或者其他能挂起线程的命令），挂起就会造成中断，并抛出 InterruptedException 异常。因此，能造成阻塞的方法（如 wait）就会抛出这个异常，所以必须在一个声明了能够抛出这个异常的方法

中调用这些方法，或者将这些方法的调用封装在一个 try-catch 块中。因为 Thread 类和 Runnable 接口的 run 方法不抛出这个异常，所以必须用 try-catch 块来封装（无论是直接的还是间接的）线程对阻塞方法的调用。但这对主线程无效，因为 main 方法可以抛出 InterruptedException 异常。

线程的中断状态可用于说明线程应当终止。为了实现这个功能，线程的 run 方法应当设置为周期性地检测（使用 isInterrupted 方法或者 interrupted 方法）线程中断状态；如果线程已经中断，线程应当以旧方式从它的 run 方法中返回。如果线程在等待的过程中中断，那么 InterruptedException 异常应该被捕获，并使用处理程序确保线程正确终止。在大部分并行程序中，都不需要对线程进行外部终止，InterruptedException 异常的 catch 块要么用于提供调试信息，要么不进行任何操作。

引入 Callable 接口以替代 Runnable 接口，该接口允许抛出异常（另外，像前面已经讨论过的，它还允许返回结果）。同时，这个接口支持泛型类型。

305
↓
306

# 术 语 表

**Abstract Data Type，ADT（抽象数据类型）** 由一些值和在这些值上的可用操作所组成的数据类型。这些值和操作独立于特定的值的表示或者操作的实现。在那些直接支持 ADT 的语言中，这些数据类型的操作在接口中体现，而这些操作的实现对用户来说是不透明的。原则上，这些实现也可以在不影响使用类型的客户的前提下修改。关于 ADT 最经典的例子就是栈（栈是有它的操作定义的），典型的栈包括 `push` 和 `pop` 操作。栈内部实现有可能会非常不同。

**abstraction（抽象）** 这个概念在不同的上下文中有不同的含义。在软件中，它通常意味着组合一个小型操作或者数据集合，并对其命名。比如，控制抽象就把一组操作结合进一个过程，然后给这个过程命名。再举一个例子，在面向对象编程中类就是一个包含数据和控制的抽象。更加一般地说，一个抽象是捕获实体的基本特性的一个表示，但是它隐藏了特定的细节。我们将经常谈论一个已经命名的抽象而不关心它们的实际细节，因为细节可能是不确定的。

**adress space（地址空间）** 进程或者处理器可以访问的地址范围。根据情况，这里要么指物理内存要么指虚拟内存。

**ADT** 参考 abstract data type。

**Amdahl's law（Amdahl 定律）** 阐述最大加速比的定律，在某些条件下（参见 2.5 章节），在一个有 $P$ 个处理器的系统上运行某个算法能得到的最大加速是：

$$S(P) = \frac{T(1)}{(\gamma + \frac{1-\gamma}{P})T(1)} = \frac{1}{\gamma + \frac{1-\gamma}{P}}$$

式中，$\gamma$ 是一个程序的串行部分，$T(n)$ 是运行在 $n$ 个处理器上的总执行时间，参考 speedup 和 serial fraction。

**AND parallelism（“与”并行性）** 一种将并行性引入逻辑语言的主要技术。考虑目标 `A:B,C,D`（读作：A 跟随着 B 和 C 和 D），意思是当且仅当次级目标 `B`、`C`、`D` 都成功目标 `A` 才成功。在“与”并行性中，次级目标 `B`、`C`、`D` 使用并行方法进行求值。

**API** 参考 application programming interface。

**API（应用程序编程接口）** 定义了为使用某个软件模块所提供的服务，另一个软件模块（通常是应用程序）所需要使用的调用约定和其他信息。MPI 就是并行编程的 API。有时候这个术语可能会被程序员不严格地用来定义表示程序中的某个特定功能的符号。比如，OpenMP 规范称为 API。API 的一个重要方面是，使用它编写的程序可以在支持这种 API 的系统上重新编译并运行。

**atomic（原子性）** 原子性在不同上下文中有细微差别。硬件等级上的一个原子操作是不可中断的，如 load、store，以及 test-and-set 指令。在数据库领域，一个原子操作（或者称作事务）是一种要么完全执行，要么完全不执行的操作。在并行编程中，原子操作是提供了足够同步且不会被其他 UE 打断的操作。原子操作也必须保证能够终止（比如，没有无限循环）。

**autoboxing（自动装箱）** 在 Java 2 1.5 中可以使用的一种语言特性，它提供了从基本数据类型到对应封装类型的原子性的自动转换，比如，从 `int` 转换为 `Integer`。

**bandwidth（带宽）** 系统的性能之一，通常用表达系统每秒所处理的条目。在并行计算中，带宽常常表

示每秒可以通过网络链接的字节数。一个会产生少量大消息的并行程序可能会因为网络的带宽而被限制，在这种情况下，这种程序称为带宽受限程序。参考 bisection bandwidth。

**barrier（栅栏）** 在一组 UE 上使用的一种同步机制，直到所有 UE 都达到栅栏后才有 UE 会继续执行。换句话说，UE 达到栅栏后将被挂起或者阻塞，当所有 UE 到达后，它们将继续运行。

**Beowulf cluster（Beowulf 集群）** 一个由运行 Linux 操作系统的 PC 组成的集群。当 Beowulf 集群在 20 世纪 90 年代首次建立起来之前，集群已经很好地发展起来了，但是这些早于 Beowulf 集群的集群运行的是 UNIX 系统。Beowulf 集群通过降低集群的硬件成本，极大增加了集群应用。

**bisection bandwidth（对分带宽）** 一些节点的两个等大小分区之间的网络双向容量。这个网络是在每个对分网络的最窄点进行划分的。

**broadcast（广播）** 把一条消息传给一个接收组内所有接收者，通常所有 UE 都会参与计算。

**cache（缓存）** 一个相对较小、访存比计算机的主存快得多的内存区域。因为处理器性能要比计算机的主存高很多，所以由一级或者多级缓存组成的缓存架构在现代操作系统中是必需的。当数据在使用前载入缓存，同时这些数据将会在计算过程中使用多次时，处理器将会高速运行。数据以缓存行形式（大小以字节计）在缓存和计算机主存之间移动。当访问内存映射到缓存的任何一个字节时，整个缓存行就会移动。当缓存被使用完毕，并且其他数据需要空间或者数据要被某些其他处理器访问时，缓存行会根据某种协议从缓存中移除。通常情况下，每个处理器都有自己的缓存（尽管多处理器会共享某一级缓存），所以保持缓存一致性（即，保证所有的处理器看到的内存内容是一致的）是一个计算机架构师和编译器作者必须解决的问题。程序员在优化软件性能时必须意识到缓存问题。

**ccNUMA** 缓存一致性 NUMA。数据在各级缓存中都具有一致性。参考 NUMA。

**cluster（集群）** 集群是指连接起来用做并行计算机的不同计算机的任何集合，或者用来形容为了高可靠性而构造的一个冗余系统。集群中的计算机没有特别为集群计算和云设计，从原则上讲，这些计算机可以独立用作单独的计算机。换句话说，组成集群的组件（包括计算机和连接它们的网络）并没有为集群而定制。这样的例子包括以太网连接的工作站网络和安装在机架上用于并行计算的工作站。参考 workstation farm。

**collective communication（集合通信）** 需要一组 UE 完成的高级操作，它的核心任务是在 UE 间协作交换信息。这种高级操作可能只是单纯的通信事件（比如，广播）或者这项工作也涉及计算（比如，归约）。参考 broadcast、reduction。

**concurrent execution（并发执行）** 指两个或者多个 UE 同时有效并同时进行的情况。这些 UE 要么同时工作在不同 PE 上，要么在同一 PE 上交替进行。

**concurrent program（并发程序）** 有多个控制轨迹（线程、进程等）的程序。

**condition variable（条件变量）** 条件变量是监控同步机制的一部分。在监控条件满足某些特定条件前，条件变量用来延迟进程或者线程的运行。当条件变成真时，它也用来唤醒一个延迟的进程。与每个条件变量相关联的是一组挂起的（延迟的）等待进程或线程。对于一个条件变量，可以实施的操作包括 `wait`（把特定进程或者线程加入等待这个变量的集合中）和 `signal` 或者 `notify`（唤醒等待集合中的进程或者线程）。参考 monitor。

**copy on write（写时复制）** 一种使用最少量同步操作并确保并发线程看到一致数据结构的技术。在这种技术中，为了更新数据结构，首先需要复制，修改针对复制项进行；然后指向旧结构的引用被自动替换为一个指向新结构的引用。这就意味着一个拥有持有指向旧结构引用的线程会继续读以前的版本（这个版本是一致的），但没有线程会看到结构的不一致状态。只有当获取和更新结构引用时才需要同步，以进行串行更新。

**counting semaphore（统计信号量）** 一种可以用任意整数表示其状态的信号量。一些统计信号量通过 `P` 和 `V` 操作完成对状态（以原子方式）递增或递减一个整数。参考 semaphore。

**cyclic distribution（周期分配法）** 一种针对数据（比如，数组中的元素）或者任务（比如，迭代）的分配方法，这种分配方法将集合划分成比 UE 数量多的小块，接着把这些块循环像分发扑克一样分配给 UE。

**data parallel（数据并行）** 数据并行是一类并行计算。这种类型的并行计算把并发描述为使用一条指令流同时操作数据结构的多个元素。

**deadlock（死锁）** 并行编程中的一种常见错误条件，其意思是计算因 UE 互相堵塞而停止执行，多个 UE 相互等待对方执行完成而阻塞。

**design pattern（设计模式）** 一种“在特定上下文中问题的解决方案”。它代表一种对设计中重复发生问题的高质量的解决方案。

**distributed computing（分布式计算）** 一种把计算任务分割成若干子任务并在多台计算机上执行的计算方式。实现分布式计算的网络一般是通用网络（LAN、WAN 或者 Internet）链接的集群，而不是使用专用网络互连的集群。

**DSM（分布式共享内存）** 由不同且分布式的内存组成的内存子系统，并可被多个 UE 共享。存在很多支持分布式共享内存系统的操作系统和硬件，或者使用软件作为中间层也可实现分布式共享内存。参考 Virtual shared memory。

**DSM** 参考 distributed shared memory。

**eager evaluation（热情计算）** 一种调度策略，这种策略要求在判断表达式或者程序的所有参数后（而不是之前），马上开始执行。热情计算对于大多数编程环境是非常典型的计算方式，与之相对的是惰性计算。当一个参数不会在计算中真正使用但必须计算时，热情计算会导致额外工作（这些工作有时甚至是无法终结的）。

**efficiency（效率）** 加速比除以 *PE* 的数目（*P*）所得到的值。它的定义如下：

$$E(P) = \frac{S(P)}{P}$$

其用于表明并行计算机中资源的使用情况。

**embarrassingly parallel（易并行）** 任务间完全独立的一种任务并行算法。参考 task parallelism 模式。

**explicitly parallel language（显式并行语言）** 指程序员可以完全定义并发性以及在并行计算中如何实现这种并发性的一种编程语言。OpenMP、Java 和 MPI 就是显式并行语言。

**factory（工厂）** 一种具有创建对象的方法的类，它通常是一个抽象基类的任何一个子类的实例。工厂类的设计模式（*抽象工厂和工厂方法*）在 [GHJV95] 中给出。

**false sharing（伪共享）** 当两个语义上不依赖的变量被放置在同一缓存行，并且运行在多个处理器中 UE 修改它们时会导致伪共享。发生伪共享的变量是语义独立的，所以它们在内存中没有冲突，但是保存变量的缓存行会在处理器间不停移动，因此会严重影响性能。

**fork（派生）** 参考 fark/join。

**fork/join（派生 / 聚合）** 派生 / 聚合是一个用于多线程 API（如 OpenMP）的编程模型。一个线程执行 fork 操作派生新线程，这些线程（在 OpenMP 中称为*线程组*）并发执行。当线程组的成员完成其任务后，执行 join 操作并被挂起，直到线程组的所有成员都执行到 join 操作。此时，新线程被销毁，原始线程继续运行。

**framework（框架）** 一个可重用、体现了特定领域的应用程序设计、部分完成的程序。程序员可通过提供应用程序专用的组件来完成程序。

**future variable（未来变量）** 在某些并行环境中用于协调 UE 执行的机制。**未来变量**是一种保存异步计算最后结果的专用变量。比如，Java（在 `java.util.concurrent` 包中）就包含 `Future` 类以保存未来变量。

**generics（泛型）** 一种编程语言特性，可以为某些特定实体（一般是数据类型）保留占位符。泛型组件的定义要在泛型使用前完成。Ada、C++ 和 Java 中都有泛型。

**grid（网格）** 一种分布式计算和资源共享结构。一个网格系统由一系列被局域网或广域网（通常是互联网）连接的异构资源集合构成。这些独立的资源都是平常使用的计算资源，包括计算服务器、存储、应用服务器、信息服务，甚至科研设施。网格通常由 Web 服务实现，并且集成对网格提供一致接口的中间件。

网格和集群不同，网格中的资源不由单个管理节点控制，而是由网格中间件管理整个系统，因此网格上各种资源的控制和资源域的使用策略由资源拥有者决策。

heterogeneous（**异构**） 由多种不同组件构成的系统。比如，一个由不同类型处理器构成的分布式系统。

homogeneous（**同构**） 由相同类型的组件组成的系统。

hypercube（**超立方**） 一种把节点放置在 $d$ 维立方体顶点的多计算机系统。最常见的配置是二进制超立方，共有 $2^n$ 个节点，每一个节点都和其他 $n$ 个节点相连。

implicitly parallel language（**隐式并行语言**） 程序并行区域的判断以及并发性的实现都由编译器实现的并行编程语言。大多数并行功能和数据流语言是隐式并行语言。

incremental parallelism（**递增并行性**） 一种将现有系统并行化的技术。这项技术作为递增更改序列将并行化引入系统，一次并行化一个循环。紧随每次改变的是对程序的详细测试，以保证程序行为和原始程序相同，这样极大地减少了引入未知 bug 的机会。参考 refactoring。

Jave Virtual Machine，JVM（**Java 虚拟机**） 一个基于栈的抽象计算机，其指令为 Java 字节码。通常，Java 程序被编译成包含 Java 字节码、符号表和其他信息的类文件。JVM 的目的是为类文件提供一个不需要关注底层平台的统一执行环境。

join（**派生**） 参考 fork/join。

JVM 参考 Java Virtual Machine。

latency(**延迟**） 响应一个请求的固定成本，比如，发送一条消息或者从磁盘上读取信息。在并行计算中，这个词经常用来描述在通信介质中发送一条空消息所需要的时间，即从调用发送例程到空消息被接收方接收所需要的时间。如果程序产生大量小消息，那么将会对延迟非常敏感。因此，这类程序也称作*延迟敏感程序*。

lazy evaluation（**惰性求值**） 一种调度策略，即在表达式（或者调用过程）的计算结果使用之前，不会计算这些内容。惰性求值可避免不必要的工作，并在有些情况下会终止一个正常情况下不会终止的计算。惰性求值经常用来进行功能编程或者逻辑编程。

Linda 并行编程的一种协同语言。参考 tuple space。

load balance（**负载均衡**） 在并行计算中，把任务交给 UE，然后再映射到 PE 上执行。PE 集合所执行的工作是与该计算相关的“负载”。负载均衡就是如何将负载尽可能平均分配到 PE 上。一个高效的并行程序中，负载是均衡的，这样每个 PE 的计算时间将大致相同。换句话说，一个具有良好负载均衡的程序中，每个 PE 完成任务的时间大致相同。

load balancing（**负载均衡方法**） 负载均衡方法是指将任务分配到 UE 中，使得每个参加并行计算的 UE 完成任务的时间大致相等。有两种负载均衡的方法：*静态负载均衡*，即任务分配在计算开始之前就已经完成；*动态分配*，即负载量会随着计算的进行而改变，任务分配在运行时进行。

locality（**局部性**） 指 PE 使用与其相关联的数据（离该 PE 近的数据）完成计算任务。比如，对于很多稠密线性代数问题，获得高性能的关键是将矩阵拆分成小块，并使用这些小块进行计算。这样被载入内存缓存中的数据就能重用。这是一个改变算法从而提高局部性的例子。

Massively Parallel Processor，MPP（**大规模并行处理器**） 一种设计规模至少为几百个处理器的分布式内存并行计算机。为了使计算机具有更好的可扩展性，MPP 机器的计算单元或者节点会定制的。这通常包含计算单元和可扩展网络的紧密集成。

Message Passing Interface，MPI（**消息传递接口**） 一种被大多数 MPP 厂商和集群计算社区采用的标准消息传递接口。该接口的存在极大增加了程序的可移植性，在一个平台上开发的基于 MPI 的程序也应该能在任何有 MPI 实现的机器上运行。

Multiple Instruction，Multiple Data，MIMD（**多指令流多数据流**） Flynn 分类法中计算机架构的一种。在 MIMD 系统中，每个 PE 有自己的指令流去操作其自己的数据。大量的现代并行系统使用 MIMD 架构。

monitor（**监控器**） 一个由 Hoare [Hoa74] 提出的同步机制。一个监控器是一个 ADT 实现，这样能够保

证在获取数据的时候互斥。条件同步在条件变量中有描述。参考 condition variable。

MPI 参考 Message Passing Interface。

MPP 参考 massively parallel processor。

multicomputer（**多计算机**） 基于 MIMD 和分布式内存并行架构的并行计算机。从用户角度看，整个系统就好像是一台计算机。

multiprocessor（**多处理器**） 多个处理器共享地址空间的并行计算机。

mutex（**互斥体**） 互斥体是一种互斥锁，可以串行化多线程执行。

node（**节点**） 常用于描述组成分布式内存并行计算机的计算单元。每个节点有自己的存储器和至少一个处理器，也就是说，节点既可以是单处理器也可以是某种多处理器。

NUMA 一个用于描述共享内存计算机系统的术语，在这种计算机系统中，内存的访问距离对于所有处理器来说不是均等的。因此，访问内存的不同位置所需要的时间是不一致的。程序员为了获得较高性能通常需要关注数据在内存中的位置。

opaque type（**不透明类型**） 一种可以在不需要知道其内部表示方式的情况下使用的类型。不透明类型的实例都可以通过已定义的接口创建和使用。MPI 消息域和 OpenMP 的锁就是不透明数据类型的典型例子。

OpenMP 一种为表示 Fortran 和 C/C++ 程序的共享内存并行性，而定义编译制导指令、库例程和环境变量的编程规范。OpenMP 可以在很多平台上实现。

OR parallelism（**“或”并行性**） 并行逻辑语言中的一种执行技术。通过使用这种技术，多个从句可并行执行。比如，一个问题有两个从句：`A:B,C` 和 `A:E,F`。这些从句能并行执行，直到其中一个成功。

parallel file system（**并行文件系统**） 对系统中任意一个处理器可见并可被多 UE 同时读写的文件系统。尽管并行文件系统在整个计算机系统看来是单个文件系统，但是它实际上分布在多个磁盘上。另外，并行文件系统的读写操作的总吞吐量必须是可扩展的。

parallel overhead（**并行开销**） 在并行计算中，花费在并行管理而不是在计算上的时间。其中包括线程创建、调度、通信以及同步。

PE 参考 processing element。

peer-to-peer computing（**端到端计算**） 各个节点地位都相同的一种分布式计算模型。在这种术语所描述的最典型的计算中，每个节点都提供相同的功能，同时任何节点可以启动一个与其他节点的会话通信。与之相对的，比如 CS 计算模型。端到端计算中共享的功能包括计算及文件共享。

POSIX 由 IEEE 计算机协会的可移植应用程序标准协会（PASC）定义的可移植操作系统接口。虽然其他操作系统也使用 POSIX 的一些标准，但这个术语主要指的是为 UNIX 和类 UNIX（如 Linux）操作系统定义的接口标准。

precedence graph（**优先级图**） 一种表示一组语句顺序约束的方法。优先级图中的节点代表语句，如果语句 *A* 要在语句 *B* 前执行，*A* 到 *B* 间就会画上一条有向边。优先级图如果有一个环则代表其中有死锁，不能够执行。

process（**进程**） 使得程序指令得以运行的资源集合。这些资源包括虚拟内存、I/O 描述符、运行时栈、信号处理程序、用户和组 ID 以及访问控制令牌。如果从一个更高的角度看进程，进程就是一个有自己地址空间的“重量级”UE。参考 unit of execution、thread。

process migration（**进程迁移**） 在进程执行时改变运行它的处理器。进程迁移在动态负载平衡的多处理器系统中经常使用。通过将进程从失败的处理器中移除，进程迁移也用来支撑容错系统。

Processing Element，PE（**处理单元**） 用于形容执行一条指令流的硬件元素的通用术语。哪个硬件单元可以定义为 PE 通常由上下文指定。考虑一个集群 SMP 工作站，在某些编程环境中，认为每个工作站都执行了一个指令流，这种情况下，PE 就是工作站。但是，当不同的编程环境运行在同一个硬件上时，可能就会把工作站的每个处理器作为执行指令流的对象，在这种情况下，PE 就是指的是处理器而不是工作站。

**programming environment（编程环境）** 提供构建程序所需要的基本工具和 API。编程环境暗指一个称作*编程模型*的计算机系统的特定抽象。

**programming model（编程模型）** 计算机系统的抽象，比如，传统串行计算机所使用的冯·诺依曼模型。对于并行计算，有很多反映处理器互连方式的不同编程模型。最常见的编程模型或者基于共享内存，或者基于分布式内存（使用消息传递），或者基于两者的结合。

**Pthread** POSIX Thread 的另一个名字，也就是，在不同 POSIX 标准中对线程的定义。参考 POSIX。

**Parallel Virtual Machine，PVM（并行虚拟机）** 一个用于并行计算的消息传递库。PVM 在并行计算的历史中扮演了非常重要的角色，因为它是第一个可移植的消息传递环境，并在并行计算社区中得到了广泛的使用。然而，现在它已经几乎被 MPI 所取代。

**race condition（竞态条件）** 这是并行程序特有的一种错误条件，程序会随着 UE 调度的变化而产生不同的结果。

**read/writer lock（读 / 写锁）** 类似于互斥锁的一对锁，多个 UE 可以拥有读锁，但是写锁与所有读锁和其他写锁互斥。当锁保护的资源读比写更加频繁的时候，读 / 写锁会更加有效率。

**reduction（归约）** 一种操作，获取对象（通常从每一个 UE 上获取一个对象）的集合，并将它们组合为位于一个 UE 上的对象，或者使每一个 UE 都拥有组合对象的一个副本。归约通常使用一种符合结合律和交换律的运算符对对象集合进行两两组合，例如，加法或者求最大值。

**refactoring（重构）** 一项软件工程技术。仔细重建程序，达到改变内部结构但不改变外部表现的目的。重构通过一系列小的改变（称为*重构*）完成，会验证每个小的改变它是否保留原有行为。这样，每次改变完成之后，整个系统可以完整工作，极大减少了引入严重、未检测 bug 的机会。递增式并行性可以看做并行编程的一种应用程序重构方式。参考*递增并行性*。

**Remote Procedure Call，RPC（远程过程调用）** 一个过程在不同的地址空间中被调用，经常发生在不同的机器上。远程过程调用是进程间通信和在分布式 CS 计算环境下启动远程进程的一种非常流行的方法。

**RPC** 参考 Remote Procedure Call。

**semaphore（信号量）** 一种用于实现特定同步的 ADT。一个信号量包括一个非负整数数值和两个原子操作。它可以进行的操作是 V（有时称作 up）和 P（有时称作 down）。一个 V 操作为信号量加 1，一个 P 操作为信号量减一（假设这个操作不违反信号量非负这一限制）。当信号量变为 0 时，已经启动的 P 操作会被挂起，当信号量变为正数时，操作可能会继续进行。

**serial fraction（串行比）** 大多数计算是由可并行部分和一定要串行执行的部分组成的。*串行比*是程序必须串行执行部分所占用的时间与整个程序运行总时间的比值。比如，如果一个程序可分解为 setup、compute 和 finalization 三部分，那么有

$$T_{total} = T_{setup} + T_{compute} + T_{finalization}$$

如果 setup 和 finalization 阶段必须串行执行，那么串行比为：

$$\gamma = \frac{T_{setup} + T_{finalization}}{T_{total}}$$

**shared address space（共享地址空间）** 被很多 UE 共享的可编址内存块。

**shared memory（共享内存）** 由硬件和软件定义的可被多个系统部件所共享的内存区域。对于编程环境而言，这个术语表明内存被多个进程或者线程所共享。对于硬件来说，它意味着将处理器连接在一起的架构特性是共享内存。参考 shared address space。

**shared nothing（无共享）** 除 LAN 外，节点间没有任何共享的分布式内存 MIMD 系统。

**Simultaneous MultiThreading，SMT（同步多线程）** 一种处理器架构特性，允许多线程在每个周期中发出指令。换句话说，SMT 允许组成处理器的功能单元可同时处理多个线程。比如，采用英特尔公司“Hyper-Threading”技术的微处理器就是 SMT 的典型代表。

**Single Instruction，Multiple Data，SIMD（单指令多数据）** 计算机体系结构 Flynn 分类法中的一种。在 SIMD 系统中，一个单独的指令流运行在多个处理器上，每个处理器都有其自己的数据流。

**single-assignment variable（单赋值变量）** 一种只可以赋值一次的变量。变量一开始处于为非赋值状态，一旦赋值后，就无法改变它。这些变量通常用于那些采用数据流控制策略的编程环境，只有所有出入变量都被赋值后，任务才开始执行。

**SMP** 参考 Symmetric multiprocessor。

**SMT** 参考 simultaneous multithreading。

**speedup（加速比）** 指并行程序性能比其串行版本高多少倍。加速比 $S$ 可表示为：

$$S(P) = \frac{T(1)}{T(P)}$$

式中，$T(n)$ 表示程序在具有 $n$ 个 PE 的系统上的总执行时间。当并行计算中加速比和 PE 数量相同时，这个加速称为完美线性。

**Single Program，Multiple Data，SPMD（单程序多数据）** 最常见的并行程序组织方式，特别是在 MIMD 计算机中。把单个程序写入并装载进并行计算机的不同节点上独立运行（暂且不论协同事件），每个节点上的指令流可以完全不同。代码中不同路径的选择依赖于节点 ID。

**SPMD** 参考 single program，multiple data。

**stride（内存跨度）** 一个结构在内存中遍历时的增量。内存跨度精确的含义随着上下文而定。比如，一个 $M*N$ 的数组存储在一个以列为主序的连续内存块中，当按列遍历时，跨度为一，当按行遍历时，跨度为 $M$。

**Symmetric MultiProcessor，SMP（对称多处理器）** 一种共享内存计算机，其每个处理器在功能上是相同的，访问每个内存地址所需要的时间也是相同的。换句话说，内存地址和操作系统服务对每个处理器均是可用的。

**synchronization（同步）** 指在不同 UE 上发生事件顺序的强制限制。这主要用于确保 UE 集访问共享资源的共享方式，这种方式使 UE 无论如何调度都能保证程序的正确性。

**systolic array（脉动阵列）** 由处理器阵列所组成的一种并行架构，其中每个处理器都和其附近的少量邻居相连。数据流通过整个阵列，当数据到达一个处理器时，该处理器执行指定的操作并且将结果传递给它的一个或者多个最近邻居。尽管脉动阵列中的每个处理器能够执行不同的指令流，但是它们以锁步（lock-step）的方式在计算和通信阶段转换。因此脉动阵列和 SIMD 架构非常类似。

**systolic algorithm（脉动算法）** 一种并行算法，使用规则的最近邻居通信模式同步任务操作。许多计算问题可以通过重新组织为某种特定类型的递归关系而转换为脉动算法。

**task（任务）** 一起工作的指令序列，对应于算法或者程序的某些逻辑部分。

**task queue（任务队列）** 一个包含被一个或多个 UE 所执行的任务的队列。任务队列一般在使用任务并行模式的程序中用来实现动态调度算法，特别是当使用主 / 从模式的时候。

**thread（线程）** 线程是在特定计算机上的基本执行单位。在 UNIX 中，线程与进程相关联并共享进程的环境。这使得线程是轻量级的（也就是说，线程的调度代价很小）。从更高视角来说，线程是与其他线程共享地址空间的轻量级执行单元。参考 unit of execution、process。

**transputer（晶片机）** 由 Inmos 公司开发的并且集成了支持并行处理芯片的微处理器。每个处理器有 4 个高速通信链路，这使得它们非常容易和其他晶片机连接，同时它也有非常高效的内置调度器。

**tuple space（元组空间）** 一个共享内存系统，其中内存中的元素被组织成称为元组的复合对象。一个元素是一小组字段，可以保存值或变量，如下所示：

(3, "the larch", 4)

(*X*, 47, [2,4,89,3])

("done")

就像我们在示例中看到的，组成元组的字段可以保存整数、字符串、变量、数组或者其他一些在基本编程语言中定义的值。传统的内存系统通过地址访问对象，元组则通过相关性。操作元组的程序员定义一个模板，并且要求系统传递匹配模板的元素。元组空间作为 Linda 协调语言的一部分创建[CG91]。Linda 语言很小，只有少量原语，用来插入元组、移除元组和提取一个元组的副本。Lina 与基本编程语言（如 C/C++ 或 Fortran）结合，创造了一个混合并行编程语言。除了在共享内存机器上的原始实现外，Linda 在虚拟共享内存的机器上也有实现，并用来实现运行在分布式内存计算机不同节点上的 UE 间的通信。这种由 Linda 启发的虚拟共享内存思想已经被纳入到了 JavaSpace[FHA99] 中。

UE 参考 unit of execution。

Unit of Execution，UE（**执行单元**） 用来表示并发执行的实体集合中的一个元素的通用术语，通常用来表示进程或者线程。参考 process、thread。

vector supercomputer（**向量超级计算机**） 将向量硬件单元集成到中央处理单元极上的超级计算机。向量硬件用流水线方式处理数组。

virtual shared memory（**虚拟共享内存**） 提供共享内存抽象的系统，即便底层硬件是基于分布式内存架构的，也允许程序员编写共享内存程序。虚拟共享内存系统可以在操作系统实现，也可以作为编程环境中的一部分。

workstation farm（**工作站农场**） 一个由通常运行 UNIX 操作系统的工作站组成的集群。在有些情况下，“农场”暗指整个系统用来运行大量独立的串行作业而不是并行计算。

# 参考文献


[ABE+97] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. PLAPACK: Parallel linear algebra package design overview. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pages 1–16. ACM Press, 1997.

[ABKP03] Mark F. Adams, Harun H. Bayraldar, Tony M. Keaveny, and Panayiotis Papadopoulos. Applications of algebraic multigrid to large-scale finite element analysis of whole bone micro-mechanics on the IBM SP. In *Proceedings SC'03*. IEEE Press, 2003. Also available at `http://www.sc-conference.org/sc2003`.

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, pages 1–6. ACM Press, June 1990.

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[ADE+01] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEC OMP: A new benchmark suite for measuring parallel computer performance. In *OpenMP Shared Memory Parallel Programming*, volume 2104 of *Lecture Notes in Computer Science*, pages 1–10. Springer-Verlag, 2001.

[AE03] Vishal Aslot and Rudolf Eigenmann. Quantitative performance analysis of the SPEC OMP 2001 benchmarks. *Scientific Programming*, 11:105–124, 2003.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[AJMJS02] J. Al-Jaroodi, N. Mohamed, H. Jiang, and D. Swanson. A comparative study of parallel and distributed Java projects for heterogeneous systems. In *Proceedings of the IPDPS'02 4th International Workshop on Java for Parallel and Distributed Computing (JavaPDC2002)*. IEEE, 2002.

[AML+99] E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez, and N. Navarro. Exploiting multiple levels of parallelism in OpenMP: a case study. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 172–180. IEEE Computer Society, 1999.

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[ARv03] C. Addison, Y. Ren, and M. van Waveren. OpenMP issues arising in the development of parallel BLAS and LAPACK libraries. *Scientific Programming*, 11(2):95–104, 2003.

[BB99] Christian Brunschen and Mats Brorsson. OdinMP/CCP: A portable implementation of OpenMP for C. In *Proceedings of the European Workshop on OpenMP*. 1999. Also available at `http://www.community.org/eayguade/resPub/papers/ewomp99/brunschen.pdf`.

[BBC+03] Christian Bell, Dan Bonachea, Yannick Cote, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Michael Welcome, and Katherine A. Yelick. An evaluation of current high-performance networks. In *Proceedings of the 17th IPDPS*. IEEE, 2003.

[BBE+99] Steve Bova, Clay Beshears, Rudolf Eigenmann, Henry Gabb, Greg Gaertner, Bob Kuhn, Bill Magro, Stefano Salvini, and Veer Vatsa. Combining message-passing and directives in parallel applications. *SIAM*, 32(9), November 1999.

[BC87] K. Beck and W. Cunningham. Using pattern languages for object-oriented programs. Presented at Workshop on Specification and Design, held in connection with OOPSLA 1987. Also available at `http://c2.com/doc/oopsla87.html`.

[BC00] M. A. Baker and D. B. Carpenter. A proposed Jini infrastructure to support a Java message passing implementation. In *Proceedings of the 2nd Annual Workshop on Active Middleware Services*. Kluwer Academic Publishers, 2000. Held at HPDC-9.

[BCC+97] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1997.

[BCKL98] Mark Baker, Bryan Carpenter, Sung Hoon Ko, and Xinying Li. mpiJava: A Java interface to MPI. Presented at First UK Workshop on Java for High Performance Network Computing, Europar 1998. Available at `http://www.hpjava.org/papers/mpiJava/mpiJava.pdf`, 1998.

[BCM+91] R. Bjornson, N. Carriero, T. G. Mattson, D. Kaminsky, and A. Sherman. Experience with Linda. Technical Report RR-866, Yale University Computer Science Department, August 1991.

[BDK95] A. Baratloo, P. Dasgupta, and Z. M. Kedem. CALYPSO: a novel software system for fault-tolerant parallel processing on distributed platforms. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*. IEEE, 1995.

[Beo] Beowulf.org: The Beowulf cluster site. `http://www.beowulf.org`.

[BGMS98] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. `http://www.mcs.anl.gov/petsc`, 1998.

[BH86] Josh Barnes and Piet Hut. A hierarchical O(N log N) force calculation algorithm. *Nature*, 324(4), December 1986.

[BJK+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1): 55–69, August 1996.

[BKS91] R. Bjornson, C. Kolb, and A. Sherman. Ray tracing with network Linda. *SIAM News*, 24(1), January 1991.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* John Wiley & Sons, 1996.

[BP99] Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical Report, University of Texas, 1999. See also `http://www.cs.utexas.edu/users/hood/`.

[BT89] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation Numerical Methods.* Prentice Hall, 1989.

[But97] David R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1st edition, 1997.

[CD97] A. Cleary and J. Dongarra. Implementation in ScaLAPACK of divide-and-conquer algorithms for banded and tridiagonal linear systems. Technical Report CS-97-358, University of Tennesee, Knoxville, 1997. Also available as LAPACK Working Note #124 from `http://www.netlib.org/lapack/lawns/`.

[CDK+00] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP.* Morgan Kaufmann Publishers, 2000.

[Cen] The Center for Programming Models for Scalable Parallel Computing. `http://www.pmodels.org`.

[CG86] K. L. Clark and S. Gregory. PARLOG: Parallel programming in logic. *ACM Trans. Programming Language Systems,* 8(1):1–49, 1986.

[CG91] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course.* MIT Press, 1991.

[CGMS94] N. J. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing,* 20:633–655, 1994.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Toward a realistic model of parallel computation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pages 1–12. May 1993.

[CLL+99] James Cowie, Hongbo Liu, Jason Liu, David M. Nicol, and Andrew T. Ogielski. Towards realistic million-node Internet simulations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 1999).* CSREA Press, 1999. See also `http://www.ssfnet.org`.

[CLW+00] A. Choudhary, W. Liao, D. Weiner, P. Varshney, R. Linderman, and R. Brown. Design, implementation, and evaluation of parallel pipelined STAP on parallel computers. *IEEE Transactions on Aerospace and Electronic Systems,* 36(2):528–548, April 2000.

[Co] Co-Array Fortran. `http://www.co-array.org`.

[Con] Concurrent ML. `http://cml.cs.uchicago.edu`.

[COR] CORBA FAQ. `http://www.omg.org/gettingstarted/corbafaq.htm`.

[CPP01] Barbara Chapman, Amit Patil, and Achal Prabhakar. Performance-oriented programming for NUMA architectures. In R. Eigenmann and M. J. Voss, editors, *Proceedings of WOMPAT 2001 (LNCS 2104)*, pages 137–154. Springer-Verlag, 2001.

[CS95] J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[DD97] J. J. Dongarra and T. Dunigan. Message-passing performance of various computers. *Concurrency: Practice and Experience*, 9(10):915–926, 1997.

[DFF+02] Jack Dongarra, Ian Foster, Geoffrey Fox, Ken Kennedy, Andy White, Linda Torczon, and William Gropp, editors. *The Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers, 2002.

[DFP+94] S. Das, R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp system for shared memory multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, pages 1332–1339. Society for Computer Simulation International, 1994.

[DGO+94] P. Dinda, T. Gross, D. O'Hallaron, E. Segall, J. Stichnoth, J. Subhlok, J. Webb, and B. Yang. The CMU task parallel program suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, March 1994.

[DKK90] Jack Dongarra, Alan H. Karp, and David J. Kuck. 1989 Gordon Bell prize. *IEEE Software*, 7(3):100–104, 110, 1990.

[Dou86] A. Douady. Julia sets and the Mandelbrot set. In H.-O. Peitgen and D. H. Richter, editors, *The Beauty of Fractals: Images of Complex Dynamical Systems*, page 161. Springer-Verlag, 1986.

[DS80] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1), August 1980.

[DS87] J. J. Dongarra and D. C. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM J. Sci. and Stat. Comp.*, 8:S139–S154, 1987.

[EG88] David Eppstein and Zvi Galil. Parallel algorithmic techniques for combinatorial computation. *Annual Reviews in Computer Science*, 3:233–283, 1988.

[Ein00] David Einstein. Compaq a winner in gene race. *Forbes.com*, June 26, 2000. `http://www.forbes.com/2000/06/26/mu7.html`.

[EM] Rudolf Eigenmann and Timothy G. Mattson. OpenMP tutorial, part 2: Advanced OpenMP. Tutorial presented at SC'2001 in Denver, Colorado, USA, 2001. Available at `http://www.cise.ufl.edu/research/ParallelPatterns/sc01-omp-tut-advanced.ppt`.

[EV01] Rudolf Eigenmann and Michael J. Voss, editors. *OpenMP Shared Memory Parallel Programming*, volume 2104 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[EWO01] Selected papers from the Second European Workshop on OpenMP (EWOMP 2000). Special issue. *Scientific Programming*, 9(2–3), 2001.

[FCO90] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 12:349, 1990.

[FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces: Principles, Patterns, and Practice.* Addison-Wesley, 1999.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems.* Prentice Hall, 1988.

[FK03] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure,* 2nd edition. Morgan Kaufmann Publishers, 2003.

[FLR98] Matteo Frigo, Charles Leiserson, and Keith Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM Press, 1998.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers,* C-21(9), 1972.

[GAM+00] M. Gonzalez, E. Ayguade, X. Martorell, J. Labarta, N. Navarro, and J. Oliver. NanosCompiler: Supporting flexible multilevel parallelism in OpenMP. *Concurrency: Practice and Experience, Special Issue on OpenMP,* 12(12):1205–1218, October 2000.

[GG90] L. Greengard and W. D. Gropp. A parallel version for the fast multipole method. *Computers Math. Applic.,* 20(7), 1990.

[GGHvdG01] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft.,* 27(4):422–455, December 2001. Also see `http://www.cs.utexas.edu/users/flame/`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 3rd edition, 1996.

[Gloa] Global Arrays. `http://www.emsl.pnl.gov/docs/global/ga.html`.

[Glob] The Globus Alliance. `http://www.globus.org/`.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface,* 2nd edition. The MIT Press, 1999.

[GOS94] Thomas Gross, David R. O'Hallaron, and Jaspal Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel & Distributed Technology,* 2(3):16–26, 1994. Also see `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/iwarp/member/fx/public/www/fx.html`.

[GS98] William Gropp and Marc Snir. *MPI: The Complete Reference,* 2nd edition. MIT Press, 1998.

[Gus88] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM,* 31(5): 532–533, 1988.

[Har91] R. J. Harrison. Portable tools and applications for parallel computers. *Int. J. Quantum Chem.,* 40(6):847–863, 1991.

[HC01] Cay S. Horstmann and Gary Cornell. *Core Java 2, Volume II: Advanced Features,* 5th edition. Prentice Hall PTR, 2001.

[HC02] Cay S. Horstmann and Gary Cornell. *Core Java 2, Volume I: Fundamentals*, 6th edition. Prentice Hall PTR, 2002.

[HFR99] N. Harrison, B. Foote, and H. Rohnert, editors. *Pattern Languages of Program Design 4*. Addison-Wesley, 1999.

[HHS01] William W. Hargrove, Forrest M. Hoffman, and Thomas Sterling. The do-it-yourself supercomputer. *Scientific American*, 285(2):72–79, August 2001.

[Hil] Hillside Group. `http://hillside.net`.

[HLCZ99] Y. Charlie Hu, Honghui Lu, Alan L. Cox, and Willy Zwaenepoel. OpenMP for networks of SMPs. In *Proceedings of 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 302–310. IEEE Computer Society, 1999.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974. Also available at `http://www.acm.org/classics/feb96`.

[HPF] Paul Hudak, John Peterson, and Joseph Fasel. A Gentle Introduction to Haskell Version 98. Available at `http://www.haskell.org/tutorial`.

[HPF97] High Performance Fortran Forum: High Performance Fortran Language specification, version 2.0. `http://dacnet.rice.edu/Depts/CRPC/HPFF`, 1997.

[HPF99] Japan Association for High Performance Fortran: HPF/JA language specification, version 1.0. `http://www.hpfpc.org/jahpf/spec/jahpf-e.html`, 1999.

[HS86] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[Hud89] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[IBM02] The IBM BlueGene/L team. An overview of the BlueGene/L supercomputer. In *Proceedings of SC'2002*. 2002. `http://sc-2002.org/paperpdfs/pap.pap207.pdf`.

[IEE] IEEE. The Open Group Base Specifications, Issue 6, IEEE Std 1003.1, 2004 edition. Available at `http://www.opengroup.org/onlinepubs/009695399/toc.htm`.

[J92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[Jag96] R. Jagannathan. Dataflow models. In A. Y. H. Zomaya, editor, *Parallel and Distributed Computing Handbook*, Chapter 8. McGraw-Hill, 1996.

[Java] Java 2 Platform. `http://java.sun.com`.

[Javb] Java 2 Platform, Enterprise Edition (J2EE). `http://java.sun.com/j2ee`.

[JCS98] Glenn Judd, Mark J. Clement, and Quinn Snell. DOGMA: distributed object group metacomputing architecture. *Concurrency: Practice and Experience* 10(11–13):977–983, 1998.

[Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[JSRa] JSR 133: Java memory model and thread specification revision. `http://www.jcp.org/en/jsr/detail?id=133`.

[JSRb] JSR 166: Concurrency utilities. `http://www.jcp.org/en/jsr/detail?id=166`.

[JSRc] Concurrency JSR-166 interest site. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`.

[KLK+03] Seung Jo Kim, Chang Sung Lee, Jeong Ho Kim, Minsu Joh, and Sangsan Lee. IPSAP: A high-performance parallel finite element code for large-scale structural analysis based on domain-wise multifrontal technique. In *Proceedings SC'03*. IEEE Press, 2003. Also available at `http://www.sc-conference.org/sc2003`.

[LAM] LAM/MPI parallel computing. `http://www.lam-mpi.org/`.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LDSH95] Hans Lischka, Holger Dachsel, Ron Shepard, and Robert J. Harrison. The parallelization of a general ab initio multireference configuration interaction program: The COLUMBUS program system. In T. G. Mattson, editor, *Parallel Computing in Computational Chemistry, ACS Symposium Series 592*, pages 75–83. American Chemical Society, 1995.

[Lea] Doug Lea. Overview of package `util.concurrent`. `http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html`.

[Lea00a] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*, 2nd edition. Addison-Wesley, 2000.

[Lea00b] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM Press, 2000.

[LK98] Micheal Ljungberg and M. A. King, editors. *Monte Carlo Calculations in Nuclear Medicine: Applications in Diagnostic Imaging*. Institute of Physics Publishing, 1998.

[Man] Manta: Fast parallel Java. `http://www.cs.vu.nl/manta/`.

[Mas97] M. Mascagni. Some methods of parallel pseudorandom number generation. In Michael T. Heath, Abhiram Ranade, and Robert S. Schreiber, editors, *Algorithms for Parallel Processing*, volume 105 of *IMA Volumes in Mathematics and Its Applications*, pages 277–288. Springer-Verlag, 1997.

[Mat87] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

[Mat94] T. G. Mattson. The efficiency of Linda for general purpose scientific programming. *Scientific Programming*, 3:61–71, 1994.

[Mat95] T. G. Mattson, editor. *Parallel Computing in Computational Chemistry, ACS Symposium Series 592*. American Chemical Society, 1995.

[Mat96] T. G. Mattson. Scientific computation. In A. Zomaya, editor, *Parallel and Distributed Computing Handbook*, pages 981–1002. McGraw-Hill, 1996.

[Mat03] T. G. Mattson. How good is OpenMP? *Scientific Programming*, 11(3): 81–93, 2003.

[Mesa] MPI (Message Passing Interface) 2.0 Standard. `http://www.mpi-forum.org/docs/docs.html`.

[Mesb] Message Passing Interface Forum. `http://www.mpi-forum.org`.

[Met] Metron, Inc. SPEEDES (Synchronous Parallel Environment for Emulation and Discrete-Event Simulation). `http://www.speedes.com`.

[MHC+99] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimits, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very high resolution simulation of compressible turbulence on the IBM-SP system. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. ACM Press, 1999.

[Min97] S. Mintchev. Writing programs in JavaMPI. Technical Report MAN-CSPE-02, School of Computer Science, University of Westminster, London, UK, 1997.

[Mis86] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[MPI] MPICH—a portable implementation of MPI. `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[MPS02] W. Magro, P. Petersen, and S. Shah. Hyper-threading technology: Impact on computer-intensive workloads. *Intel Technology Journal*, 06(01), 2002.

[MR95] T. G. Mattson and G. Ravishanker. Portable molecular dynamics software for parallel computing. In T. G. Mattson, editor, *Parallel Computing in Computational Chemistry, ACS Symposium Series 592*, page 133. American Chemical Society, 1995.

[MRB97] R. C. Martin, D. Riehle, and F. Buschmann, editors. *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

[MSW96] Timothy G. Mattson, David Scott, and Stephen R. Wheat. A TeraFLOP supercomputer in 1996: the ASCI TeraFLOP system. In *Proceedings of IPPS'96, The 10th International Parallel Processing Symposium*. IEEE Computer Society, 1996.

[NA01] Dimitrios S. Nikolopoulos and Eduard Ayguadé. A study of implicit data distribution methods for OpenMP using the SPEC benchmarks. In Rudolf Eigenmann and Michael J. Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 2104 of *Lecture Notes in Computer Science*, pages 115–129. Springer-Verlag, 2001.

[NBB01] Jeffrey S. Norris, Paul G. Backes, and Eric T. Baumgartner. PTEP: The parallel telemetry processor. In *Proceedings IEEE Aerospace Conference*, volume 7, pages 7–3339 – 7–3345. IEEE, 2001. Also see `http://wits.jpl.nasa.gov:8080/WITS/publications/2001-ptep-ieee-as.pdf`.

[NHK+02] J. Nieplocha, R. J. Harrison, M. K. Kumar, B. Palmer, V. Tipparaju, and H. Trease. Combining distributed and shared memory models: Approach and evolution of the Global Arrays toolkit. In *Proceedings of Workshop on Performance Optimization for High-Level Languages and Libraries (ICS'2002)*. ACM Press, 2002.

[NHL94] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: a portable "shared-memory" programming model for distributed memory computers. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pages 340–349. ACM Press, 1994.

[NHL96] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[NM92] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[Omn] Omni OpenMP compiler. `http://phase.hpcc.jp/Omni/home.html`.

[OMP] OpenMP: Simple, portable, scalable SMP programming. `http://www.openmp.org`.

[OSG03] Ryan M. Olson, Michael W. Schmidt, and Mark S. Gordon. Enabling the efficient use of SMP clusters: the GAMESS/DDI model. In *Proceedings SC'03*. IEEE Press, 2003. Also available at `http://www.sc-conference.org/sc2003/paperpdfs/pap263.pdf`.

[Pac96] Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996.

[Pat] The Pattern Languages of Programs Conference. `http://www.hillside.net/conferences/plop.htm`.

[PH95] S. J. Plimpton and B. A. Hendrickson. Parallel molecular dynamics algorithms for simulation of molecular systems. In T. G. Mattson, editor, *Parallel Computing in Computational Chemistry, ACS Symposium Series 592*, pages 114–132. American Chemical Society, 1995.

[PH98] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, 2nd edition, 1998.

[PLA] PLAPACK: Parallel linear algebra package. `http://www.cs.utexas.edu/users/plapack`.

[Pli95] S. J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J Comp Phys*, 117(1):1–19, 1995.

[PS00] Tom Porter and Galyn Susman. On site: Creating lifelike characters in Pixar movies. *Communications of the ACM*, 43(1):25–29, 2000.

[PS04] Bill Pugh and Jaime Spacco. MPJava: high-performance message passing in Java using java.nio. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing, 16 International Workshop (LCPC 2003), Revised Papers*, volume 2958 of *Lecture Notes in Computer Science*. Springer, 2004. Also appeared in the Proceedings of MASPLAS'03. `http://www.cs.haverford.edu/masplas/masplas03-01.pdf`.

[PTV93] William H. Press, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*, 2nd edition. Cambridge University Press, 1993.

[Rep99] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[RHB03] John W. Romein, Jaap Heringa, and Henri E. Bal. A million-fold speed improvement in genomic repeats detection. In *Proceedings SC'03*. IEEE Press, 2003. `http://www.sc-conference.org/sc2003/paperpdfs/pap189.pdf`.

[RHC+96] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. Tholburn. POOMA. In *Parallel Programming Using C++*. The MIT Press, 1996.

[RMC+98] Radharamanan Radhakrishnan, Dale E. Martin, Malolan Chetlur, Dhananjai Madhava Rao, and Philip A. Wilsey. An object-oriented Time Warp simulation kernel. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*. Springer-Verlag, 1998. Also available at `http://www.ececs.uc.edu/~paw/lab/papers/warped/iscope98.ps.gz`. See also `http://www.ece.uc.edu/~paw/warped`.

[Sca] The ScaLAPACK project. `http://www.netlib.org/scalapack/`.

[Sci03] Special issue on OpenMP and its applications. *Scientific Programming*, 11(2), 2003.

[SER] Java Servlet Technology. `http://java.sun.com/products/servlet/`.

[SET] SETI@home: The search for extraterrestrial intelligence. `http://setiathome.ssl.berkeley.edu/`.

[SHPT00] S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible control structures for parallelism in OpenMP. *Concurrency: Practice and Experience*, 12:1219–1239, 2000.

[SHTS01] Mitsuhisa Sato, Motonari Hirano, Yoshio Tanaka, and Satoshi Sekiguchi. OmniRPC: A grid RPC facility for cluster and global computing in OpenMP. In Rudolf Eigenmann and Michael J. Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 2104 of *Lecture Notes in Computer Science*, pages 130–136. Springer-Verlag, 2001.

[SLGZ99] Alex Scherer, Honghui Lu, Thomas Gross, and Willy Zwaenepoel. Transparent adaptive parallelism on NOWS using OpenMP. *ACM SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 34(8):96–106, August 1999.

[SN90] Xian-He Sun and Lionel M. Ni. Another view on parallel speedup. In *Proceedings of the 1990 Conference on Supercomputing*, pages 324–333. IEEE Computer Society Press, 1990.

[SR98] Daryl A. Swade and James F. Rose. OPUS: A flexible pipeline data-processing environment. In *Proceedings of the AIAA/USU Conference on Small Satellites*. September 1998. See also `http://www.smallsat.org/proceedings/12/ssc98/2/sscii6.pdf`.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*, 2nd edition. MIT Press, 1994.

[SSD+94] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The design and evolution of Zipcode. *Parallel Computing*, 20(4):565–596, 1994.

[SSGF00] C. P. Sosa, C. Scalmani, R. Gomperts, and M. J. Frisch. Ab initio quantum chemistry on a ccNUMA architecture using OpenMP III. *Parallel Computing*, 26(7–8):843–856, July 2000.

[SSOG93] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, May 1993.

[Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 4(2):315–339, 1990.

[Tho95] John Thornley. Performance of a class of highly-parallel divide-and-conquer algorithms. Technical report, Caltech, 1995. `http://resolver.caltech.edu/CaltechCSTR:1995.cs-tr-95-10`.

[Tita] Titanium home page. `http://titanium.cs.berkeley.edu/intro.html`.

[Top] TOP500 supercomputer sites. `http://www.top500.org`.

[UPC] Unified Parallel C. `http://upc.gwu.edu`.

[VCK96] J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors. *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

[vdG97] Robert A. van de Geijn. *Using PLAPACK*. MIT Press, 1997.

[vdSD03] Aad J. van der Steen and Jack J. Dongarra. Overview of recent supercomputers. 2003. `http://www.top500.org/ORSC/`.

[VJKT00] M. Valero, K. Joe, M. Kitsuregawa, and H. Tanaka, editors. *High Performance Computing: Third International Symposium*, volume 1940 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[vRBH+98] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using Ensemble. *Software—Practice and Experience*, 28(9):963–979, August 1998. See also `http://www.cs.cornell.edu/Info/Projects/Ensemble`.

[Wie01] Frederick Wieland. Practical parallel simulation applied to aviation modeling. In *Proceedings of the Fifteenth Workshop on Parallel and Distributed Simulation*, pages 109–116. IEEE Computer Society, 2001.

[Win95] A. Windemuth. Advanced algorithms for molecular dynamics simulation: The program PMD. In T. G. Mattson, editor, *Parallel Computing in Computational Chemistry, ACS Symposium Series 592*. American Chemical Society, 1995.

[WSG95] T. L. Windus, M. W. Schmidt, and M. S. Gordon. Parallel implementation of the electronic structure code GAMESS. In T. G. Mattson, editor, *Parallel Computing in Computational Chemistry, ACS Symposium Series 592*, pages 16–28. American Chemical Society, 1995.

[WY95] Chin-Po Wen and Katherine A. Yelick. Portable runtime support for asynchronous simulation. In *Proceedings of the 1995 International Conference on Parallel Processing, Volume II: Software*. CRC Press, August 1995.

[X393] Accredited Standards Committee X3. Parallel extensions for Fortran. Technical Report X3H5/93-SDI revision M, American National Standards Institute, April 1993.

[YWC+96] Katherine A. Yelick, Chih-Po Wen, Soumen Chakrabarti, Etienne Deprit, Jeff A. Jones, and Arvind Krishnamurthy. Portable parallel irregular applications. In Takayasu Ito, Robert H. Halstead Jr., and Christian Queinnec, editors, *Parallel Symbolic Languages and Systems, International Workshop (PSLS'95)*, volume 1068 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 1996.

[ZJS+02] Hans P. Zima, Kazuki Joe, Mitsuhisa Sato, Yoshiki Seo, and Masaaki Shimasaki, editors. *Proceedings HPF International Workshop: Experiences and Progress (HiWEP 2002)*, volume 2327 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

# 索　引

索引中页码为英文原书页码，与书中页边标注的页码一致。

## C

D

## E

## F

## G

## H



## I

## M

## N



## O

## Q



## R

## S

## T

## U



## V



## W